# IGraph/M

## the igraph interface for *Mathematica*

This notebook can be opened using the command `IGDocumentation[]` or through the Documentation Centre. It cannot be saved, so feel free to edit and evaluate input cells, and experiment!

The documentation is currently incomplete. *Contributions are very welcome!*

## Introduction

IGraph/M provides a *Mathematica* interface to the popular igraph network analysis package, as well as many other functions for working with graphs in *Mathematica*. The purpose of IGraph/M is not to replace *Mathematica*'s built-in graph theory functionality, but to complement it. Thus the IGraph/M interface is designed to interoperate seamlessly with built-in functions and datatypes, while also being familiar to users of other igraph interfaces (R, Python or C).

The full igraph functionality is not yet exposed. Priority is given to functionality that is not currently built into *Mathematica*. While many of the functions that IGraph/M provides overlap with built-in ones, like `IGBetweenness` and `BetweeneessCentrality`, there are usually some relevant differences. For example, `IGBetweenness` uses edge weights, while the built-in function `BetweennessCentrality` does not.

## Basic usage

The package can be loaded using

In[1]:=
```
Needs["IGraphM`"]
```

Out[1]=
```
IGraph/M 0.6.5 (December 21, 2022)
Evaluate IGDocumentation[] to get started.
```

The list of included functions can be queried with the command below. Notice that their names always have the `IG` prefix. Click on the name of a function to see its usage message.

In[2]:=
```
? IGraphM`*
```

Or just type a question mark followed by the symbol's name:

In[3]:=
```
? IGVersion
```

IGVersion[] returns the IGraph/M version along with the version of the igraph library in use.

In[4]:=
```
IGVersion[]
```

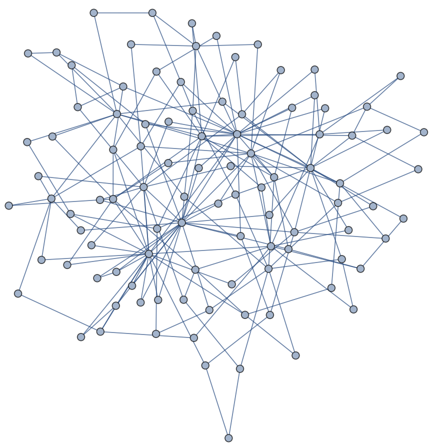
Out[4]=
```
IGraph/M 0.6.5 (December 21, 2022)
igraph 0.9.10-23-g5635203bd (Dec 21 2022)
Mac OS X x86 (64-bit)
```

IGraph/M functions work directly with *Mathematica*'s built-in `Graph` datatype. No new special graph datatype is introduced.

Let’s take a look at a few examples. Let us first generate a graph using the built-in functions of *Mathematica*.

In[5]:=
```
SeedRandom[42];
g = RandomGraph[BarabasiAlbertGraphDistribution[100, 2]]
```

Out[6]=



We can compute the betweenness centrality of each vertex either using IGraph/M, ...

In[7]:=
```
IGBetweenness[g]
```

Out[7]=
```
{1118.26, 1058.15, 540.601, 127.365, 1175.53, 678.175, 206.929, 128.576, 204.019, 535.316,
 487.858, 391.669, 0., 135.039, 0., 52.5324, 104.28, 12.2286, 75.8798, 110.155, 68.8282,
 13.9095, 46.4209, 99.3299, 0., 168.196, 213.871, 358.855, 0., 64.9572, 5.12619, 102.369,
 17.978, 15.569, 95.7266, 8.45843, 25.4984, 13.0274, 0., 71.2012, 47.2895, 32.4444, 8.20833,
 0., 27.0286, 10.9357, 4.60238, 0., 14.7095, 24.7944, 79.125, 7.38301, 22.0817, 43.9635,
 11.7135, 10.9952, 40.8782, 11.2429, 0., 60.0431, 9.36667, 32.4529, 85.4487, 100.431,
 15.205, 93.2876, 60.0548, 9.2, 0., 0., 10.512, 9.37438, 8.42222, 45.7937, 3.61667, 9.23333,
 53.3897, 11.4012, 22.0959, 5.24091, 10.2647, 8.66017, 9.97438, 11.0429, 15.8765, 12.7798,
 0., 30.1744, 0., 0., 4.0373, 9.7, 1., 10.4883, 0., 0., 13.7861, 13.8594, 1.7, 2.80952}
```

... or using *Mathematica*’s built-ins, and obtain the same result.

In[8]:=
```
BetweennessCentrality[g]
```

Out[8]=
```
{1118.26, 1058.15, 540.601, 127.365, 1175.53, 678.175, 206.929, 128.576, 204.019, 535.316,
 487.858, 391.669, 0., 135.039, 0., 52.5324, 104.28, 12.2286, 75.8798, 110.155, 68.8282,
 13.9095, 46.4209, 99.3299, 0., 168.196, 213.871, 358.855, 0., 64.9572, 5.12619, 102.369,
 17.978, 15.569, 95.7266, 8.45843, 25.4984, 13.0274, 0., 71.2012, 47.2895, 32.4444, 8.20833,
 0., 27.0286, 10.9357, 4.60238, 0., 14.7095, 24.7944, 79.125, 7.38301, 22.0817, 43.9635,
 11.7135, 10.9952, 40.8782, 11.2429, 0., 60.0431, 9.36667, 32.4529, 85.4487, 100.431,
 15.205, 93.2876, 60.0548, 9.2, 0., 0., 10.512, 9.37438, 8.42222, 45.7937, 3.61667, 9.23333,
 53.3897, 11.4012, 22.0959, 5.24091, 10.2647, 8.66017, 9.97438, 11.0429, 15.8765, 12.7798,
 0., 30.1744, 0., 0., 4.0373, 9.7, 1., 10.4883, 0., 0., 13.7861, 13.8594, 1.7, 2.80952}
```

Let us now assign weights to the edges. Many IGraph/M functions, including `IGBetweenness`, support edge weights.

In[9]:=
```
wg = SetProperty[g, EdgeWeight → RandomReal[1, EdgeCount[g]]];
```

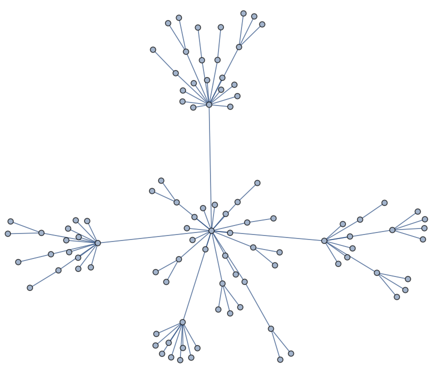
In[10]:=
```
IGBetweenness[wg]
```
Out[10]=
```
{1569., 1509., 697., 506., 1510., 948., 173., 0., 106., 827., 663., 379., 0., 318., 0., 360., 0.,
 0., 83., 129., 1., 0., 227., 582., 0., 91., 236., 213., 0., 60., 0., 334., 1., 53., 549., 0.,
 0., 0., 0., 10., 0., 0., 0., 0., 68., 68., 17., 357., 27., 16., 80., 0., 0., 0., 437., 0., 0.,
 0., 52., 22., 0., 0., 62., 139., 93., 187., 1., 7., 0., 0., 0., 16., 0., 69., 10., 98., 0., 1.,
 4., 21., 0., 0., 0., 0., 0., 0., 0., 43., 0., 0., 98., 0., 0., 0., 0., 0., 0., 63., 25., 4.}
```

Notice that *Mathematica* 13.0 does not include functionality to compute weighted vertex betweenness. The built-in function `BetweennessCentrality[]` ignores the weights.

In[11]:=
```
BetweennessCentrality[wg]
```
Out[11]=
```
{1118.26, 1058.15, 540.601, 127.365, 1175.53, 678.175, 206.929, 128.576, 204.019, 535.316,
 487.858, 391.669, 0., 135.039, 0., 52.5324, 104.28, 12.2286, 75.8798, 110.155, 68.8282,
 13.9095, 46.4209, 99.3299, 0., 168.196, 213.871, 358.855, 0., 64.9572, 5.12619, 102.369,
 17.978, 15.569, 95.7266, 8.45843, 25.4984, 13.0274, 0., 71.2012, 47.2895, 32.4444, 8.20833,
 0., 27.0286, 10.9357, 4.60238, 0., 14.7095, 24.7944, 79.125, 7.38301, 22.0817, 43.9635,
 11.7135, 10.9952, 40.8782, 11.2429, 0., 60.0431, 9.36667, 32.4529, 85.4487, 100.431,
 15.205, 93.2876, 60.0548, 9.2, 0., 0., 10.512, 9.37438, 8.42222, 45.7937, 3.61667, 9.23333,
 53.3897, 11.4012, 22.0959, 5.24091, 10.2647, 8.66017, 9.97438, 11.0429, 15.8765, 12.7798,
 0., 30.1744, 0., 0., 4.0373, 9.7, 1., 10.4883, 0., 0., 13.7861, 13.8594, 1.7, 2.80952}
```

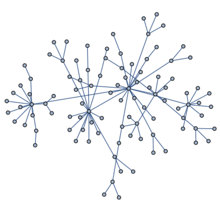
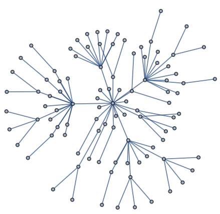
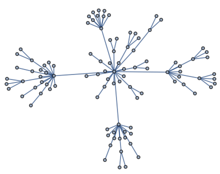
Let us delete the minimum feedback edge set to obtain an acyclic graph:

In[12]:=
```
acg = EdgeDelete[g, IGFeedbackArcSet[g]]
```
Out[12]=



And try out a few of igraph's layout algorithms.

In[13]:=
```
{IGLayoutGraphOpt[acg], IGLayoutKamadaKawai[acg], IGLayoutFruchtermanReingold[acg]}
```
Out[13]=

{, , }

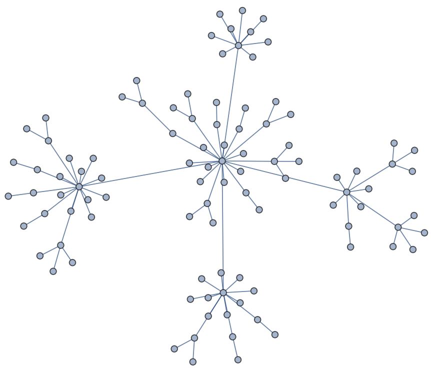
Layout functions typically have many options to tune:

In[14]:=
```
Options[IGLayoutGraphOpt]
```
Out[14]=
```
{MaxIterations → 500, NodeCharge → 0.001, NodeMass → 30, SpringLength → 0,
 SpringConstant → 1, MaxStepMovement → 5, Continue → False, Align → True}
```

Increasing the number of iterations will usually improve the result.

In[15]:=
```
IGLayoutGraphOpt[acg, "MaxIterations" → 5000]
```

Out[15]=



**A final note**

Please refer to the usage messages for information on how to use each function. For more information on the meaning of various function options, the algorithms used by the functions, references, etc. please refer to the C/igraph documentation. The igraph documentation provides article references for most nontrivial algorithms.

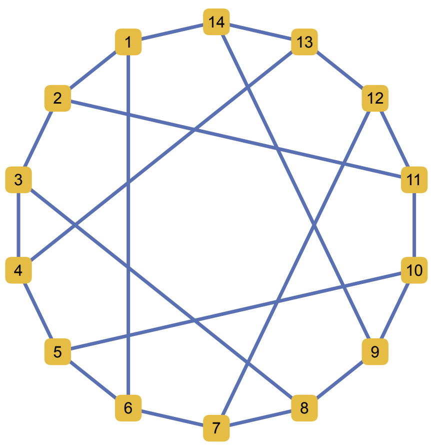
The following sections provide general information on each functionality area, and show common usage patterns.

# Graph creation

All the graph creation functions in IGraph/M take any standard *Mathematica* `Graph` option such as `VertexLabels`, `EdgeLabels`, `VertexStyle`, `GraphStyle`, `PlotTheme`, etc.

In[16]:=
```
IGLCF[{5, -5}, 7, GraphStyle → "SmallNetwork"]
```

Out[16]=




## Deterministic graph generators

### IGShorthand

`IGShorthand` provides an easy way to create small graphs from a simple and quick-to-type notation.

In[17]:=
```
? IGShorthand
```

IGShorthand["..."] builds a graph from a shorthand notation such as "a->b<-c" or "a-b,c-d".

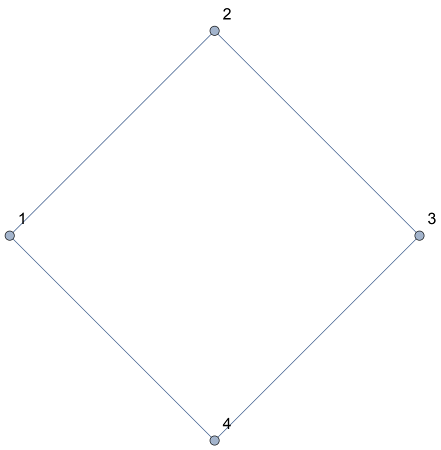
The available options are:

- SelfLoops → True keeps self-loops in the graph.
- MultiEdges → True keeps parallel edges in the graph.

Construct a cycle graph.

In[18]:=
```
IGShorthand["1-2-3-4-1"]
```

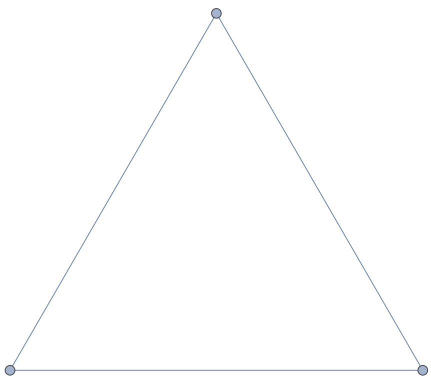
Out[18]=




Vertex labels are shown by default. They can be turned off using VertexLabels → None.

In[19]:=
```
IGShorthand["1-2-3-1", VertexLabels → None]
```

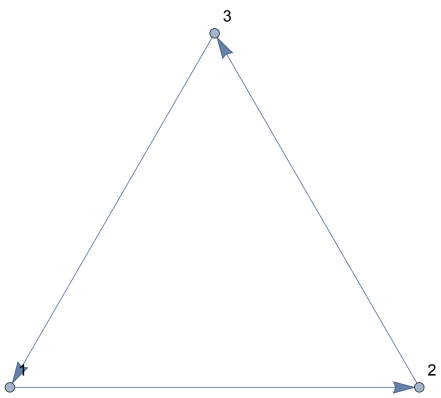
Out[19]=



The interpretation of – as directed or undirected is controlled by the DirectedEdges option.

In[20]:=
```
IGShorthand["1-2-3-1", DirectedEdges → True]
```

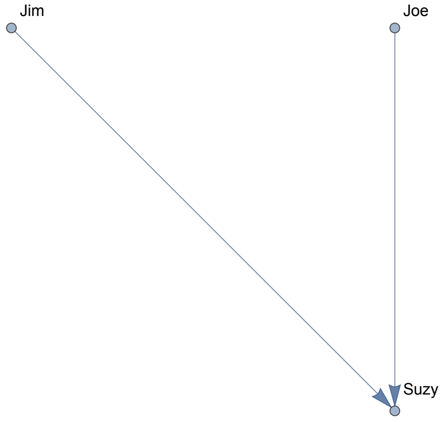
Out[20]=



Directed edges can be input using ->, <- or <->.

In[21]:=
```
IGShorthand["Jim -> Suzy <- Joe"]
```

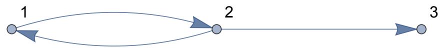
Out[21]=




<-> is interpreted as a pair of directed edges.

In[22]:=
```
IGShorthand["1<->2->3"]
```

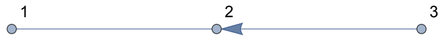
Out[22]=




Mixed graphs, containing both directed and undirected edges, are supported. Note that mixed graphs are not allowed as input to most IGraph/M functions.

In[23]:=
```
IGShorthand["1-2<-3"]
```

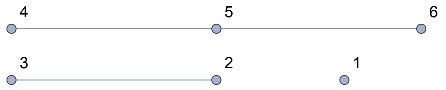
Out[23]=




Disconnected components are separated by commas.

In[24]:=
```
IGShorthand["1, 2-3, 4-5-6"]
```

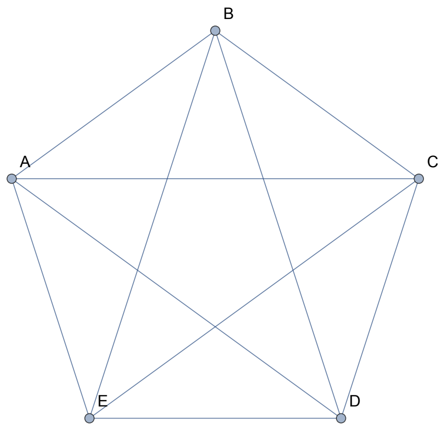
Out[24]=




Groups of vertices can be given using the colon separator.  Edges will be connected to each vertex in the group. This makes it easy to specify a complete graph ...

In[25]:=
```
IGShorthand["A:B:C:D:E -- A:B:C:D:E"]
```

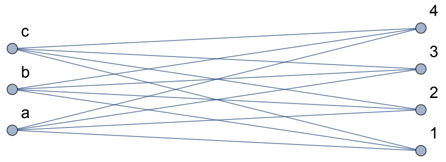
Out[25]=



... or a complete bipartite graph.

In[26]:=
```
IGLayoutBipartite@IGShorthand["a:b:c - 1:2:3:4"]
```

Out[26]=




Vertex names are taken as strings, except when they can be interpreted as an integer.

In[27]:=
```
IGShorthand["xyz - 137"] // VertexList // InputForm
```

Out[27]//InputForm=
```
{"xyz", 137}
```

Spaces are allowed in vertex names, and edges can be specified using any number of - characters.

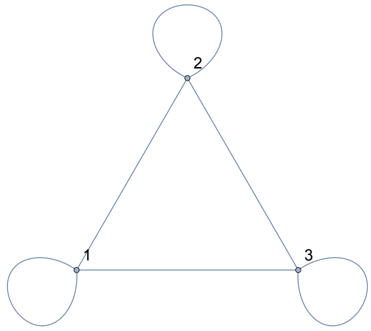
In[28]:=
```
IGShorthand["Sophus Lie --- Camille Jordan"]
```

Out[28]=


Self-loops and parallel edges are removed by default because these are often created as an undesired by-product of vertex groups. They can be re-enabled using the `SelfLoops` or `MultiEdges` options when desired.

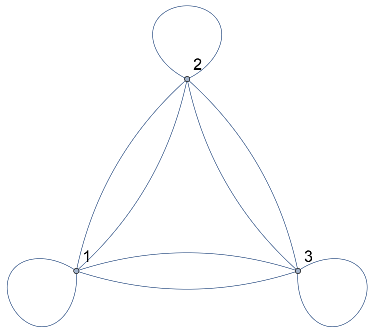
In[29]:=
```
IGShorthand["1:2:3 - 1:2:3", SelfLoops → True]
```

Out[29]=

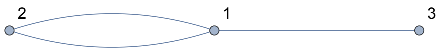



In[30]:=
```
IGShorthand["1:2:3 - 1:2:3", SelfLoops → True, MultiEdges → True]
```

Out[30]=



In[31]:=
```
IGShorthand["1-2-1-3", MultiEdges → True]
```
Out[31]=



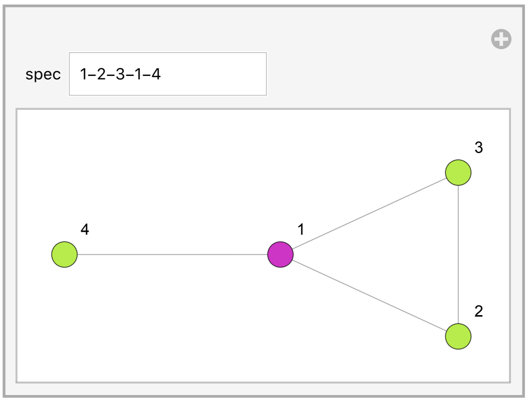

The vertex order will follow the order of appearance of vertices in the input string. To control the order, simply list vertices at the beginning of the shorthand specification.

In[32]:=
```
IGShorthand["4-3-1-2-4"] // VertexList
```
Out[32]=
```
{4, 3, 1, 2}
```

In[33]:=
```
IGShorthand["1,2,3,4, 4-3-1-2-4"] // VertexList
```
Out[33]=
```
{1, 2, 3, 4}
```

Create an interactive graph editor that dynamically visualizes betweenness.

In[34]:=
```
Manipulate[
 IGShorthand[spec, VertexSize → {"Scaled", 0.06}, EdgeStyle → Gray] //
  IGVertexMap[ColorData["NeonColors"], VertexStyle → IGBetweenness /* Rescale],
 {{spec, "1-2-3-1-4"}, InputField[#, String, ContinuousAction → True] &},
 Initialization :→ Needs["IGraphM`"]
]
```
Out[34]=




## IGEmptyGraph

In[35]:=
```
? IGEmptyGraph
```

IGEmptyGraph[] gives a graph with no edges or vertices.
IGEmptyGraph[n] gives a graph with no edges and n vertices.

`IGEmptyGraph` is a convenience function for creating graphs with no edges.

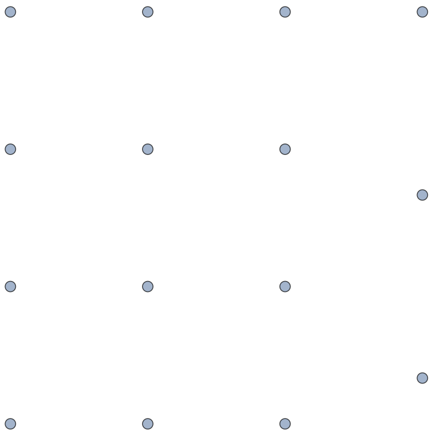
Create a null graph.

In[36]:=
```
IGEmptyGraph[] // VertexCount
```
Out[36]=
```
0
```

Create an empty graph on 15 vertices.

In[37]:=
```
IGEmptyGraph[15]
```

Out[37]=



The built-in `EmptyGraphQ` returns `True` for these graphs.

In[38]:=
```
EmptyGraphQ[%]
```

Out[38]=
```
True
```

## IGLCF

In[39]:=
```
? IGLCF
```

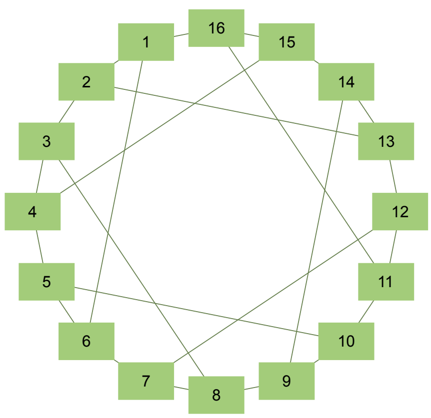
IGLCF[shifts, repeats] creates a graph from LCF notation.
IGLCF[shifts, repeats, vertexCount] creates a graph from LCF notation with the number of vertices specified.

`IGLCF[{`$k_1$`, `$k_2$`, …}, n]` creates a graph based on the LCF notation `[`$k_1$`, `$k_2$`, …]`$^n$.

The Möbius–Kantor graph is `[5, -5]`$^8$.

In[40]:=
```
IGLCF[{5, -5}, 8, GraphStyle → "DiagramGreen"]
```

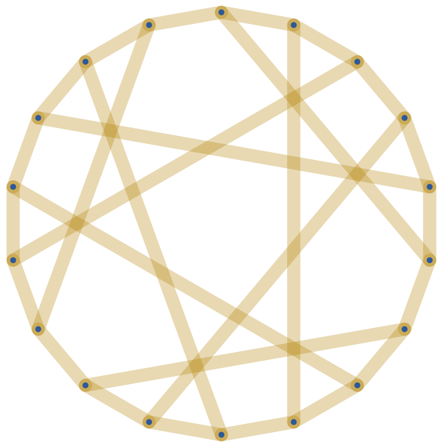
Out[40]=



The Pappus graph is `[5, 7, -7, 7, -7, -5]`$^3$.

In[41]:=
```
IGLCF[{5, 7, -7, 7, -7, -5}, 3, GraphStyle → "ThickEdge"]
```

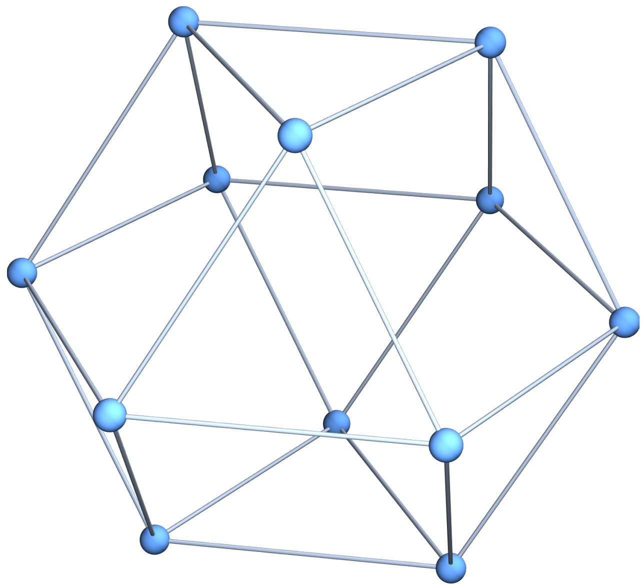
Out[41]=



The cuboctahedral graph is `[4, 2]`$^6$.

In[42]:=
```
IGLayoutKamadaKawai3D@IGLCF[{4, 2}, 6]
```

Out[42]=



## IGChordalRing

In[43]:=
```
? IGChordalRing
```

IGChordalRing[n, w] gives an extended chordal ring on n vertices, based on the vector or matrix w.

`IGChordalRing[n, w]` constructs an extended chordal ring based on the offset specification vector or matrix *w* as follows:

**1.** It creates a cycle graph (i.e. ring) on *n* vertices.

**2.** For each vertex *i* on the ring, it adds a chord to a vertex *w*⟦*i* mod *p*⟧ steps ahead counter-clockwise on the ring.

**3.** If *w* is a matrix, the procedure is carried out for each row.

The number of vertices *n* must be an integer multiple of the number of columns in the matrix *w*.

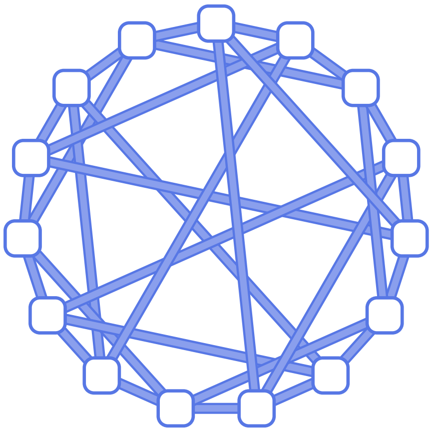
The available options are:

- `DirectedEdges` → `True` creates a graph with directed edges.
- `SelfLoops` → `False` prevents the creation of self-loops.
- `MultiEgdes` → `False` prevents the creation of multi-edges.

Create an extended chordal graph.

In[44]:=
```
IGChordalRing[15, {3, 4, 8}, GraphStyle → "Business"]
```

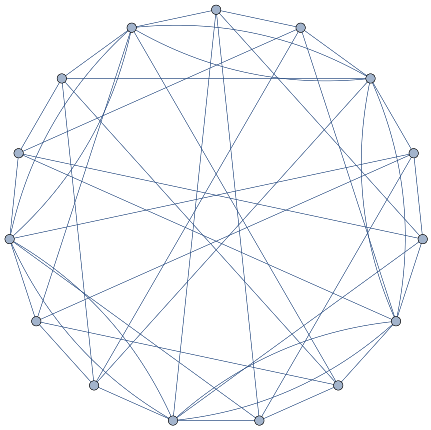
Out[44]=



Negative offsets are allowed.

In[45]:=
```
IGChordalRing[15, {{3, 4, 8}, {-3, -4, -8}}]
```

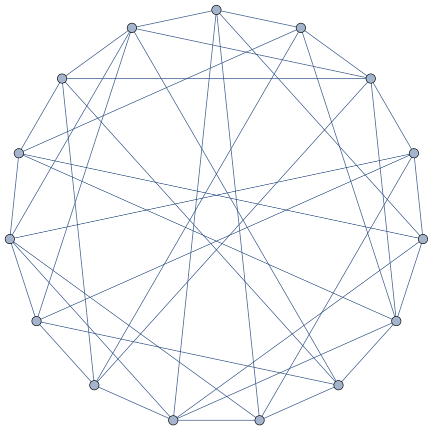
Out[45]=



`IGChordalGraph` may create self-loops or multi-edges. This can be prevented by setting the `SelfLoops` or `MultiEdges` options to `False`.

In[46]:=
```
IGChordalRing[15, {{3, 4, 8}, {-3, -4, -8}}, MultiEdges → False]
```

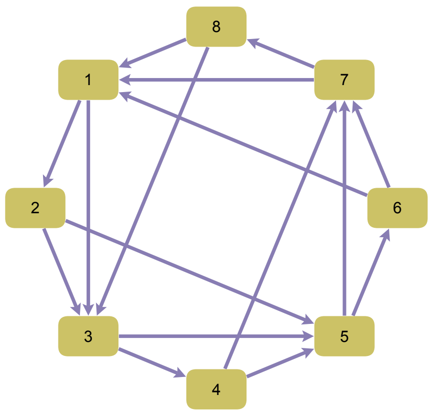
Out[46]=



Create a chordal graph with directed edges.

In[47]:=
```
IGChordalRing[8, {2, 3}, DirectedEdges → True, GraphStyle → "DiagramGold"]
```

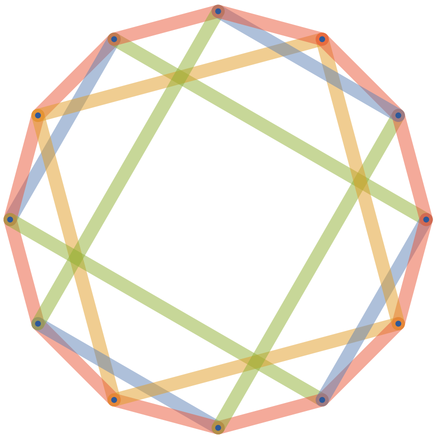
Out[47]=



Colour the chords of the ring based on which entry of the *w* vector they correspond to.

In[48]:=
```
w = {2, 3, 4};
IGChordalRing[12, w, GraphStyle → "ThickEdge", EdgeStyle → Opacity[1 / 2]] // IGEdgeMap[
  ColorData[97],
  EdgeStyle → Function[g,
    Table[If[i ≤ VertexCount[g], 0, Mod[i, Length[w], 1]], {i, EdgeCount[g]}]
   ]
 ]
```

Out[49]=



## IGSquareLattice

In[50]:=
```
? IGSquareLattice
```

IGSquareLattice[{d1, d2, …}] generates a square grid graph of the given dimensions.

`IGSquareLattice[{d1, d2, …}]` creates a square lattice graph of the given dimensions. The available options are:

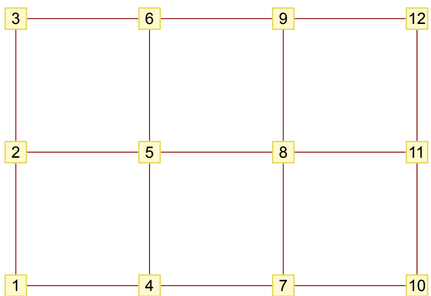
- `"Radius"` controls the size of the neighbourhood within which vertices will be connected.
- `"Periodic"` → `True` creates a periodic lattice.
- `"Mutual"` → `True` inserts directed edges in both directions when `DirectedEdges` → `True` is used.

In previous versions, `IGSquareLattice` was called `IGMakeLattice`. This name can still be used as a synonym for the sake of backwards compatibility, however, it will be removed in a future version.

To create other types of lattices, see `IGTriangleLattice` and `IGLatticeMesh`.

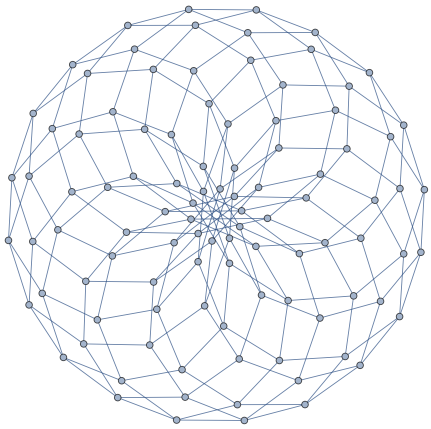
In[51]:=
```
IGSquareLattice[{3, 4}, GraphStyle → "VintageDiagram"]
```

Out[51]=



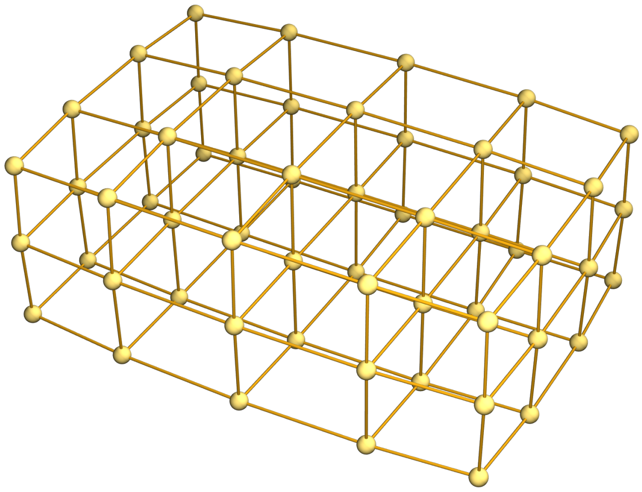
In[52]:= IGSquareLattice[{10, 10}, "Periodic" → True]

Out[52]=



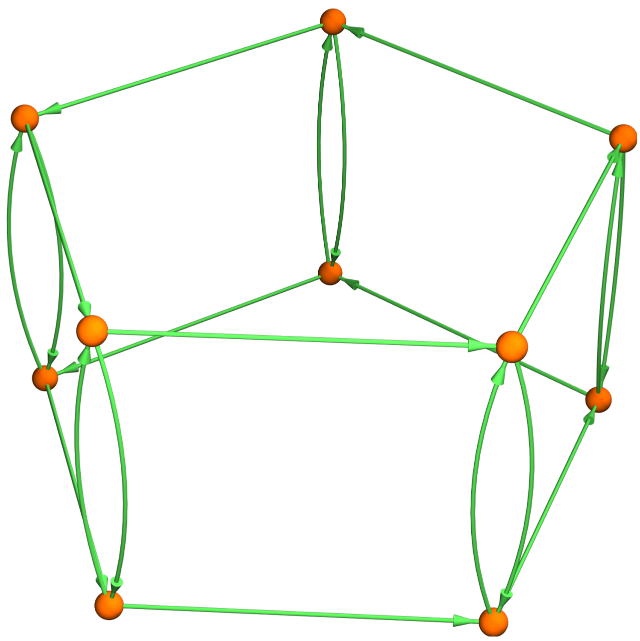
In[53]:= Graph3D@IGSquareLattice[{5, 4, 3}, GraphStyle → "Prototype"]

Out[53]=



In[54]:= Graph3D@IGSquareLattice[{2, 5}, DirectedEdges → True, "Periodic" → True, PlotTheme → "NeonColor"]

Out[54]=



## IGTriangularLattice

In[55]:=
```
? IGTriangularLattice
```

IGTriangularLattice[n] generates a triangular lattice graph on a size n equilateral triangle using n(n+1)/2 vertices.
IGTriangularLattice[{m, n}] generates a triangular lattice graph on an m by n rectangle.

`IGTriangularLattice` can create a triangular grid graph in the shape of a triangle or a rectangle. To generate other types of lattices, see `IGSquareLattice` and `IGLatticeMesh`.

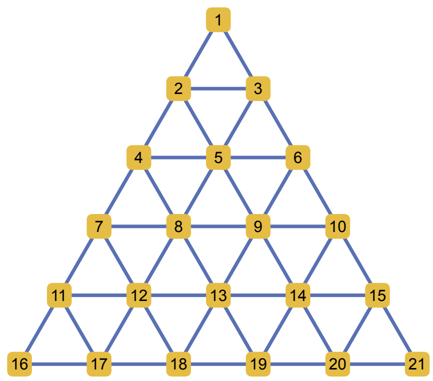
The available options are:

- `DirectedEdges → True` creates a directed graph.
- `"Periodic" → True` creates a periodic lattice.

Generate a triangular lattice on an equilateral triangle with 6 vertices along each of its edges.

In[56]:=
```
IGTriangularLattice[6, GraphStyle → "SmallNetwork"]
```

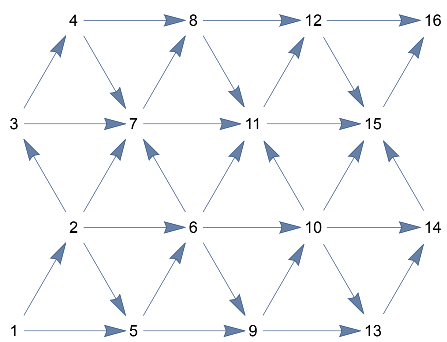
Out[56]=




Create a directed triangle lattice on a rectangle. Notice the vertex labelling and that the arrows are oriented from smaller index vertices to larger index ones, making this an acyclic graph.

In[57]:=
```
IGTriangularLattice[{4, 4}, DirectedEdges → True,
 VertexShapeFunction → "Name", PerformanceGoal → "Quality"]
```

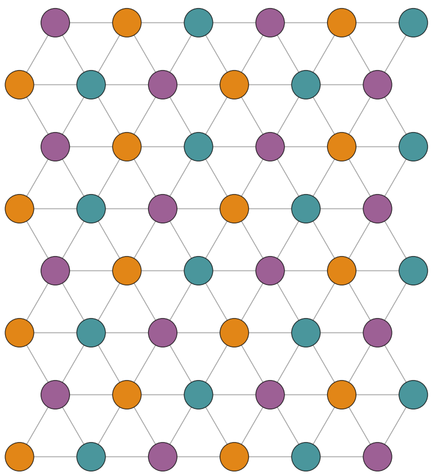
Out[57]=



Create a triangle lattice and colour its vertices.

In[58]:=
```
IGTriangularLattice[{8, 6}, VertexSize → Large, EdgeStyle → Gray] //
 IGVertexMap[ColorData[98], VertexStyle → IGMinimumVertexColoring]
```

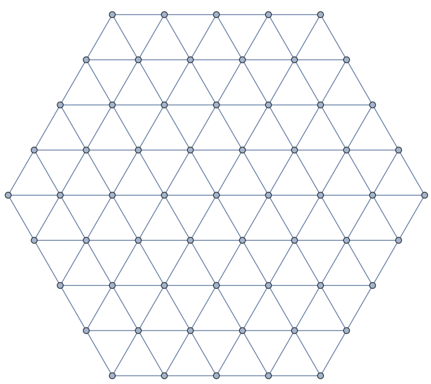
Out[58]=



Take a hexagonal subgraph of a triangle lattice.

In[59]:=
```
g = IGTriangularLattice[13];
center = First@GraphCenter[g];
VertexDelete[g,
 Complement[VertexList[g], AdjacencyList[g, center, 4], {center}]
]
```

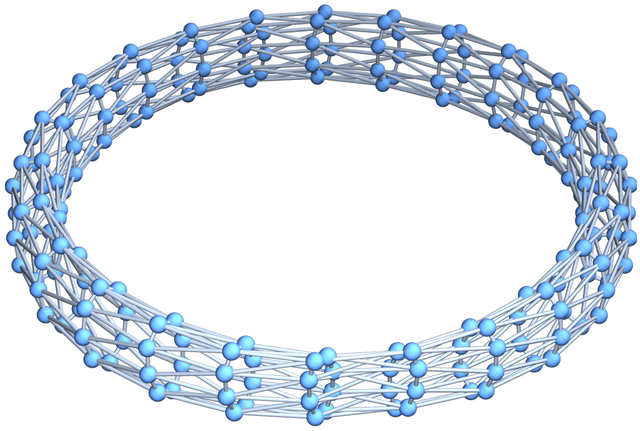
Out[61]=



Create a periodic (i.e. toroidal topology) triangle lattice.

In[62]:=
```
Graph3D@IGTriangularLattice[{24, 8}, "Periodic" → True]
```

Out[62]=



## IGKaryTree

In[63]:=
```
? IGKaryTree
```

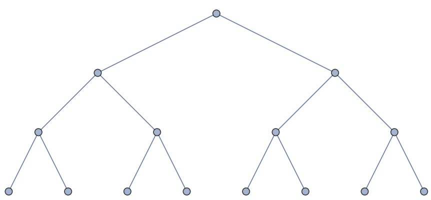
IGKaryTree[n] gives a binary tree with n vertices.
IGKaryTree[n, k] gives a k–ary tree with n vertices.

The available options are:

- `DirectedEdges → True` creates a directed tree.

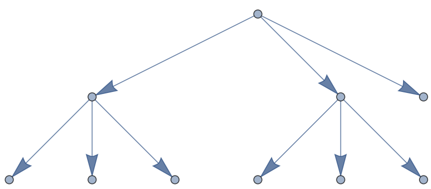
In[64]:=
```
IGKaryTree[15]
```

Out[64]=



In[65]:=
```
IGKaryTree[10, 3, DirectedEdges → True]
```

Out[65]=



## IGSymmetricTree

In[66]:=
```
? IGSymmetricTree
```

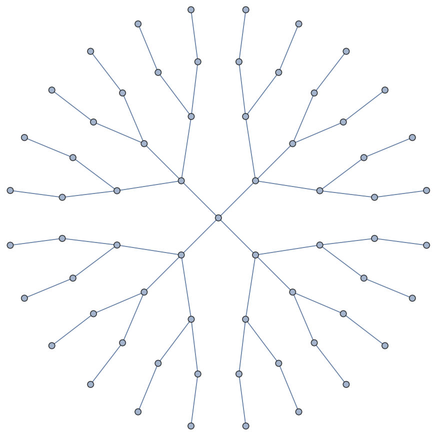
IGSymmetricTree[{k1, k2, …}] gives a tree where vertices in the (i+1)st layer have k_i children.

`IGSymmetricTree` creates a tree where successive layers (i.e. vertices at the same distance from the root) have the specified number of children.

Create a tree where the root has 4 children, its children have 3 children, and so on.

In[67]:=
```
IGSymmetricTree[{4, 3, 2, 1}]
```

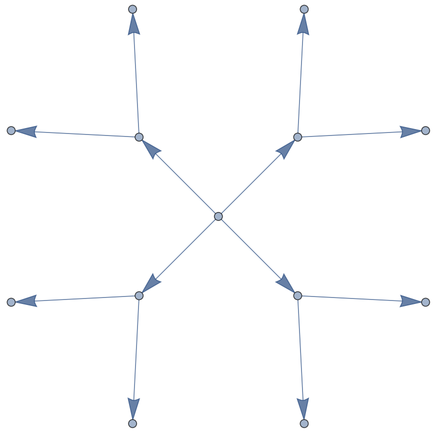
Out[67]=



Create a directed tree.

In[68]:=
```
IGSymmetricTree[{4, 2}, DirectedEdges → True]
```

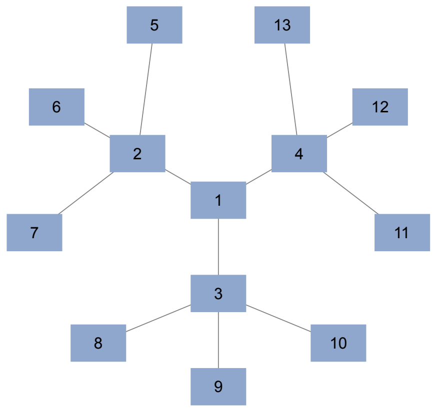
Out[68]=



`IGSymmetricTree` is guaranteed to label vertices in breadth-first order. Deeper layers have higher integer labels.

In[69]:=
```
IGSymmetricTree[{3, 3}, GraphStyle → "DiagramBlue"]
```

Out[69]=



## IGBetheLattice

In[70]:=
```
? IGBetheLattice
```

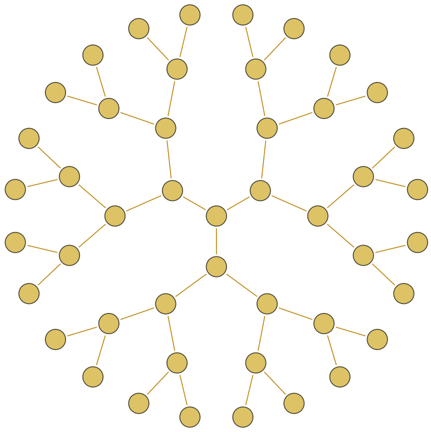
IGBetheLattice[n] gives the first n layers of a Bethe lattice with coordination number 3.
IGBetheLattice[n, k] gives the first n layers of a Bethe lattice with coordination number k.

A Bethe lattice, also called a regular tree, is an infinite tree in which all vertices have the same degree. `IGBetheLattice[n, k]` computes the first n layers of such a tree. Each non-leaf vertex will have degree k. The default degree is 3.

`IGBetheLattice` differs from `CompleteKaryTree` in that the degree of the root will be the same as the degree of other non-lead nodes.

In[71]:=
```
IGBetheLattice[5, GraphStyle → "Prototype", VertexSize → Large]
```

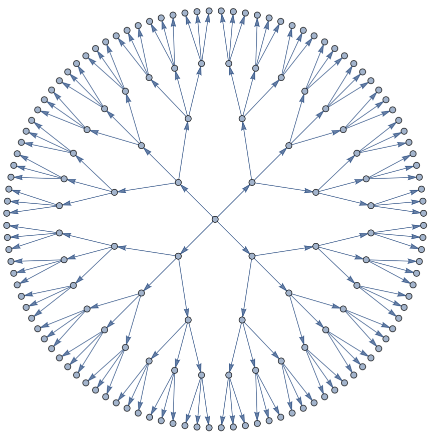
Out[71]=



Generate a tree where non-leaf nodes have total degree 5, and use directed edges.

In[72]:=
```
IGBetheLattice[5, 4, DirectedEdges → True]
```

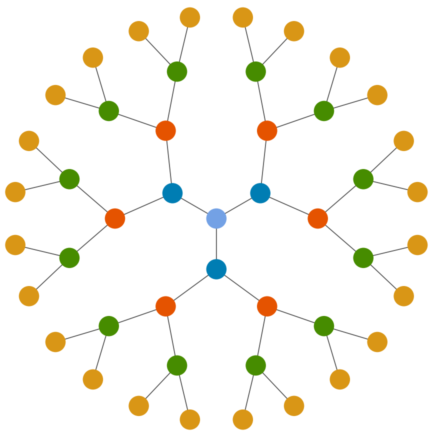
Out[72]=



Colour vertices based on their distance from the root (i.e. the “layer” they are part of).

In[73]:=
```
IGVertexMap[
 ColorData[68],
 VertexStyle → (First@IGDistanceMatrix[#, {1}] &),
 IGBetheLattice[5, GraphStyle → "BasicBlack", VertexSize → 0.4]
]
```

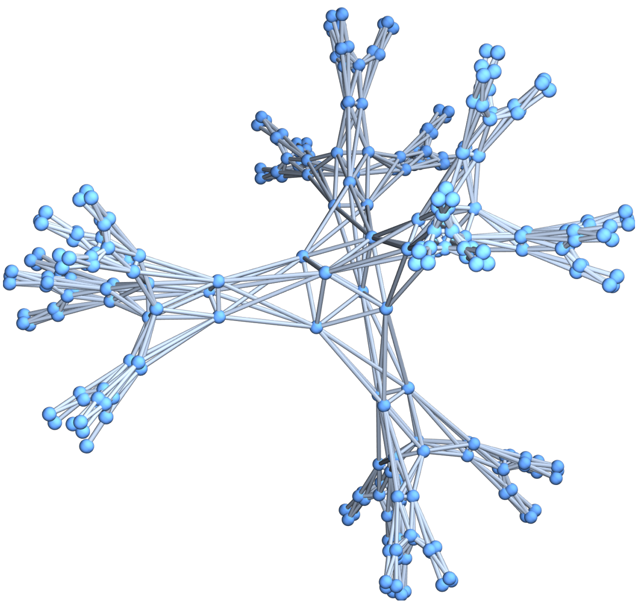
Out[73]=



Visualize the nested line graph of a degree-4 regular tree.

In[74]:=
```
Graph3D@Nest[LineGraph, IGBetheLattice[5, 4], 2]
```

Out[74]=



## IGFromPrufer

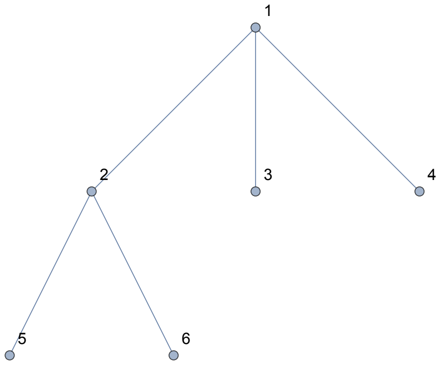
In[75]:=
```
? IGFromPrufer
```

IGFromPrufer[sequence] constructs a tree from a Prüfer sequence.

A Prüfer sequence is a unique representation of an $n$-vertex labelled tree as $n-2$ integers between 1 and $n$.

In[76]:=
```
IGFromPrufer[{1, 1, 2, 2}, VertexLabels → "Name"]
```

Out[76]=




Use `IGToPrufer` to convert a tree back to its Prüfer sequence.

In[77]:=
```
IGToPrufer[%]
```

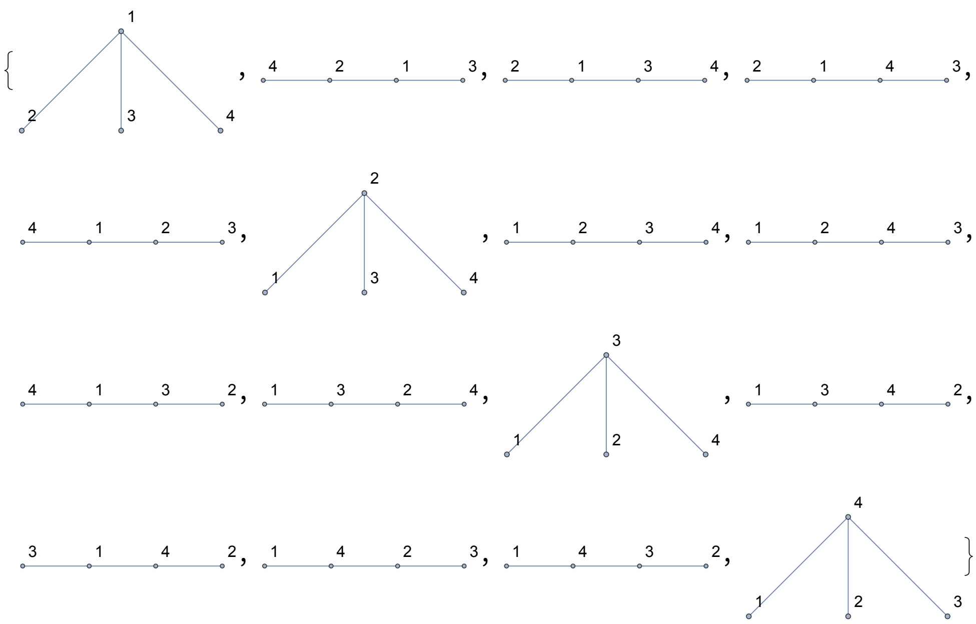
Out[77]=

```
{1, 1, 2, 2}
```

Generate all labelled trees on 4 nodes:

In[78]:=
```
IGFromPrufer[#, VertexLabels → "Name"] & /@ Tuples[Range[4], {2}]
```

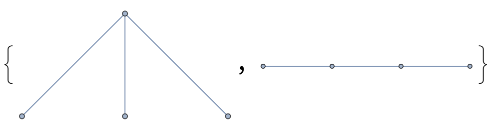
Out[78]=




Of these, only two are non-isomorphic.

In[79]:=
```
DeleteDuplicates[CanonicalGraph /@ %]
```

Out[79]=



## IGExpressionTree

In[80]:=
```
? IGExpressionTree
```

IGExpressionTree[expression] constructs a tree graph from an arbitrary Mathematica expression.

`IGExpressionTree` constructs the tree graph corresponding to an arbitrary *Mathematica* expression. The vertices of the tree will be the positions of the corresponding subexpressions.

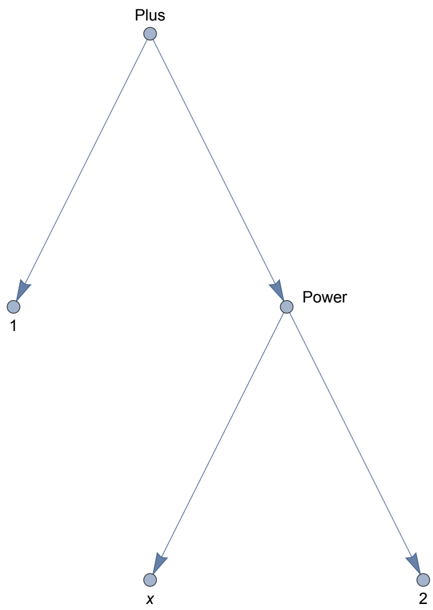
`IGExpressionTree` takes all standard `Graph` options. The `VertexLabels` option takes the following special values:

- `VertexLabels → "Head"` labels branch vertices with the `Head` of the corresponding subexpression and leaf vertices with the corresponding atomic expression.
- `VertexLabels → "Subexpression"` labels vertices with the corresponding subexpression.
- `VertexLabels → "Name"` labels vertices with their name, i.e. the position of the corresponding subexpression.
- `VertexLabels → None` uses no labels.

`IGExpressionTree` constructs a graph corresponding to the structure of a *Mathematica* expression.

In[81]:=
```
tree = IGExpressionTree[expr = 1 + x^2]
```

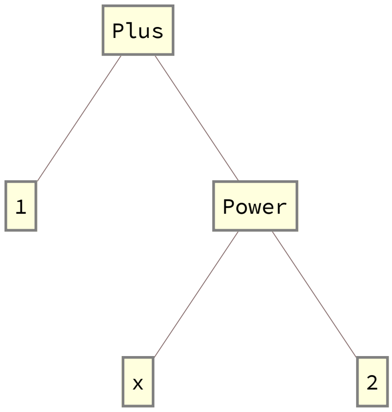
Out[81]=



The expression tree is similar to what `TreeForm` displays, but unlike `TreeForm`'s output, it is a `Graph` object that works with all graph functions.

In[82]:=
```
TreeForm[expr]
```

Out[82]//TreeForm=




The vertex names are the position specifications of the corresponding subexpressions.

In[83]:=
```
VertexList[tree]
```

Out[83]=

```
{{1}, {2, 1}, {2, 2}, {2}, {}}
```

In[84]:=
```
Extract[expr, %]
```

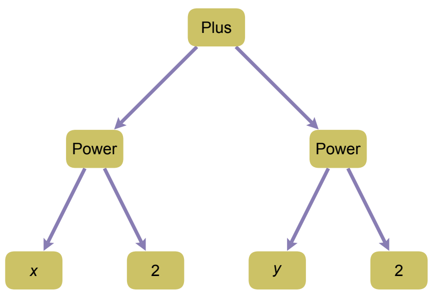
Out[84]=

$\{1, x, 2, x^2, 1 + x^2\}$

Place the vertex labels in the centre and construct an undirected graph.

In[85]:=
```
IGExpressionTree[x^2 + y^2,
 GraphStyle → "DiagramGold",
 VertexLabels → Placed["Head", Center], VertexSize → Large
]
```

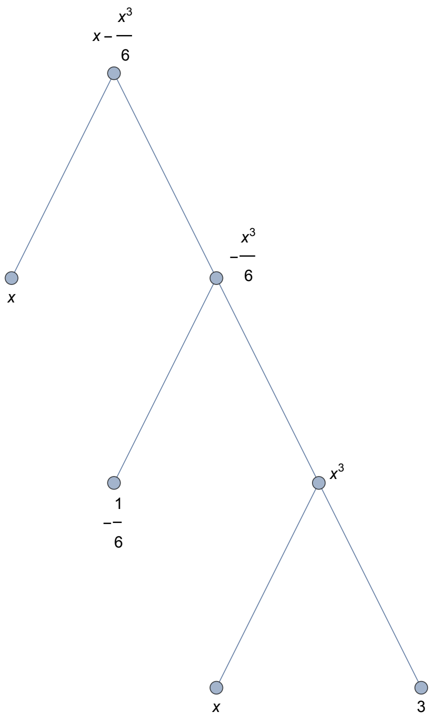
Out[85]=



Create an undirected graph, labelled with subexpressions.

In[86]:=
```
IGExpressionTree[Normal[Sin[x] + O[x]^5], DirectedEdges → False,
 VertexLabels → "Subexpression"]
```

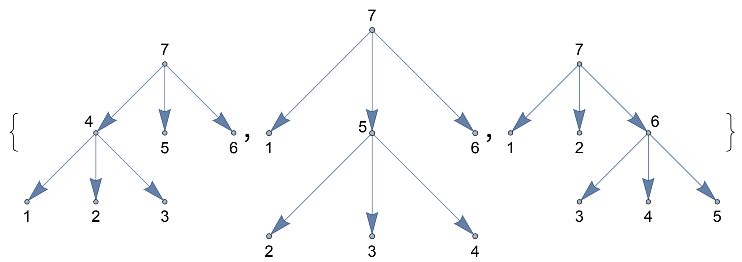
Out[86]=




Certain trees are easier to construct through their corresponding nested expression.

In[87]:=
```
IGExpressionTree[#, VertexLabels → "Index"] & /@ Groupings[5, 3]
```

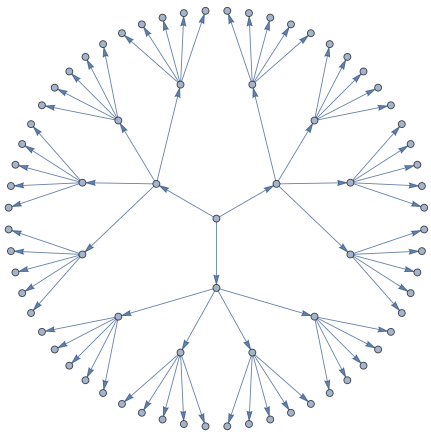
Out[87]=



An equivalent of `IGSymmetricTree` can be easily implemented using `IGExpressionTree`.

In[88]:=
```
IGExpressionTree[ConstantArray[1, {3, 4, 5}], VertexLabels → None, GraphLayout → "RadialEmbedding"]
```

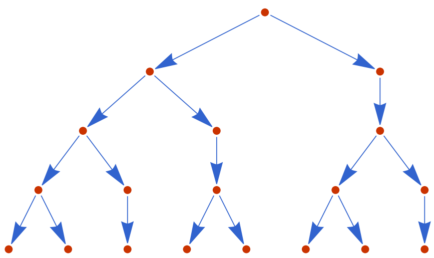
Out[88]=



Define a tree through a substitution system.

In[89]:=
```
IGExpressionTree[
 Nest[ReplaceAll[{0 → {0, 1}, 1 → {0}}], {0, 1}, 3],
 VertexLabels → None, GraphStyle → "VibrantColor"
]
```

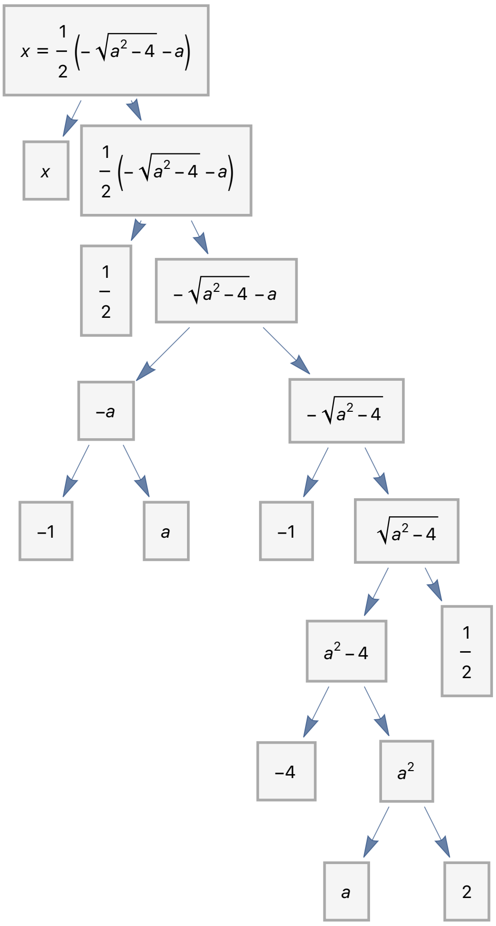
Out[89]=



To format each node so that it fits a label, it is necessary to set an explicit `VertexShapeFunction`.

In[90]:=
```
IGExpressionTree[First@Roots[x^2 + a x + 1 == 0, x],
    VertexLabels → "Subexpression",
    PerformanceGoal → "Quality",
    ImageSize → 280
  ] //
  IGVertexMap[
    Function[e, Inset[Panel[e], #1] &],
    VertexShapeFunction → IGVertexProp[VertexLabels]
  ] // RemoveProperty[#, VertexLabels] &
```

Out[90]=




## IGCompleteGraph

In[91]:=
```
? IGCompleteGraph
```

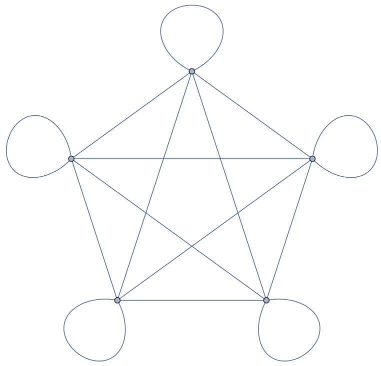
IGCompleteGraph[n] gives a complete graph on n vertices.
IGCompleteGraph[vertices] gives a complete graph on the given vertices.

The available options are:

- `DirectedEdges → True` creates a directed graph.
- `SelfLoops → True` includes self-loops.

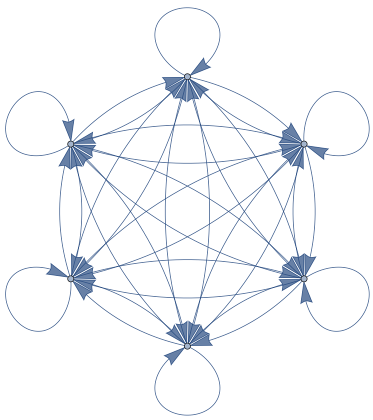
Create an undirected complete graph with loops.

In[92]:=
```
IGCompleteGraph[5, SelfLoops → True]
```
Out[92]=



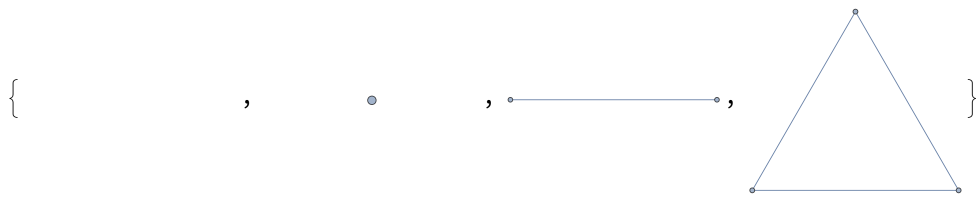
Create a directed complete graph with loops.

In[93]:=
```
IGCompleteGraph[6, SelfLoops → True, DirectedEdges → True]
```
Out[93]=



Create a list of complete graphs starting with the null graph.

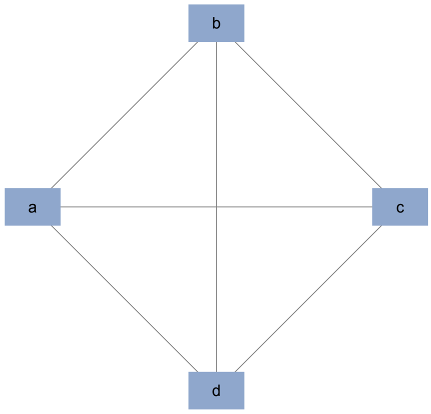
In[94]:=
```
IGCompleteGraph /@ Range[0, 3]
```
Out[94]=



Create a complete graph on the given vertices.

In[95]:=
```
IGCompleteGraph[{"a", "b", "c", "d"}, GraphStyle → "DiagramBlue"]
```

Out[95]=




## IGCompleteAcyclicGraph

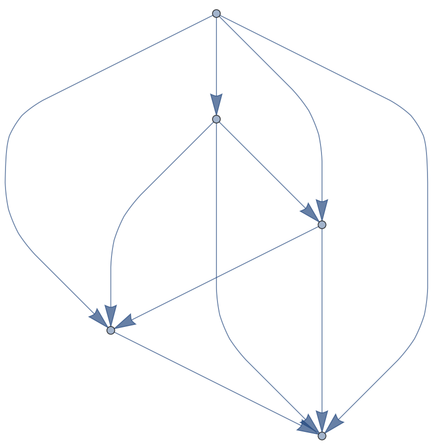
In[96]:=
```
? IGCompleteAcyclicGraph
```

IGCompleteAcyclicGraph[n] gives a complete acyclic directed graph on n vertices.
IGCompleteAcyclicGraph[vertices] gives a complete acyclic directed graph on the given vertices.

Create a complete acyclic directed graph on 5 vertices.

In[97]:=
```
IGCompleteAcyclicGraph[5]
```

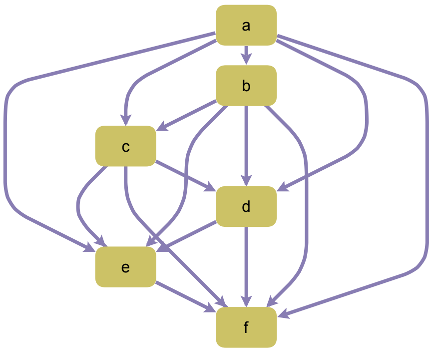
Out[97]=



Create a complete acyclic graph on the given vertices. The directed edges always run from vertices that appear earlier in the list to those that appear later.

In[98]:=
```
IGCompleteAcyclicGraph[CharacterRange["a", "f"], GraphStyle → "DiagramGold"]
```

Out[98]=




## IGKautzGraph

In[99]:=
```
? IGKautzGraph
```

IGKautzGraph[m, n] gives a Kautz graph on m+1 characters with string length n+1.

The vertices of the Kautz graph $K_m^n$ are strings of length $n+1$, composed of $m+1$ distinct symbols, with the restriction that two adjacent symbols in the string may not be the same. A vertex $s_1 s_2 \ldots s_n s_{n+1}$ connects to all other vertices of the form $s_2 \ldots s_{n+1} x$, where $x$ can be any symbol distinct from $s_{n+1}$.

The Kautz graph $K_m^n$ has $(m+1) m^n$ vertices, with each vertex having in-degree and out-degree $m$. Therefore, it has $(m+1) m^{n+1}$ edges.

In[100]:=
```
VertexCount@IGKautzGraph[3, 2] == (3 + 1) * 3^2
```

Out[100]=
```
True
```

In[101]:=
```
VertexOutDegree@IGKautzGraph[3, 2]
```

Out[101]=
```
{3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
```

The line graph of Kautz graph $K_m^n$ is $K_m^{n+1}$.

In[102]:=
```
IGIsomorphicQ[
 LineGraph@IGKautzGraph[2, 2],
 IGKautzGraph[2, 3]
]
```

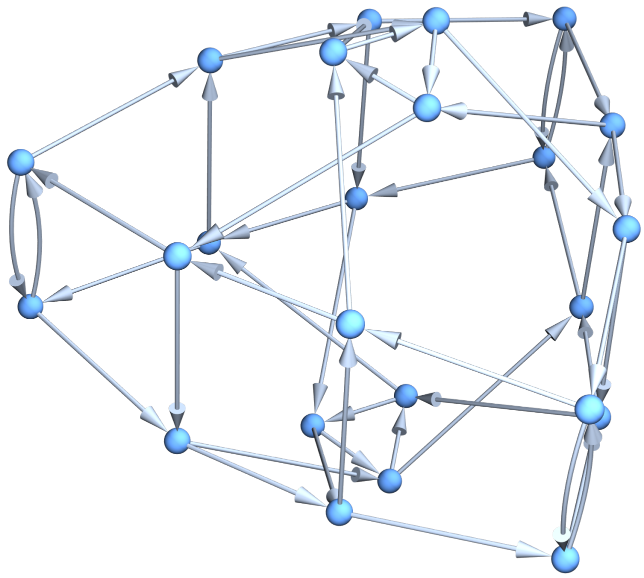
Out[102]=
```
True
```

Visualize the Kautz graph $K_2^3$ on 3 characters with string length 4 in three dimensions.

In[103]:=

```
Graph3D@IGKautzGraph[2, 3]
```

Out[103]=

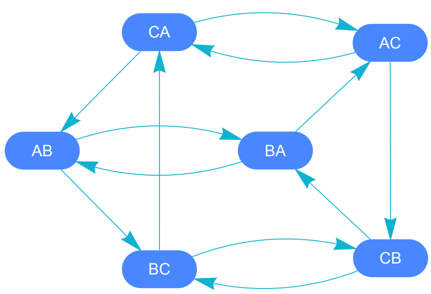


Label the vertices of the Kautz graph on 3 characters with string length 2.

In[104]:=

```
labels = StringJoin /@ DeleteCases[Tuples[{"A", "B", "C"}, {2}], {c_, c_}];
IGKautzGraph[2, 1,
 VertexLabels → Thread[Range[6] → (Placed[#, Center] &) /@ labels],
 VertexSize → Large, VertexShapeFunction → "Capsule", PerformanceGoal → "Quality",
 PlotTheme → "CoolColor", VertexLabelStyle → White
]
```

Out[105]=




## IGDeBruijnGraph

In[106]:=

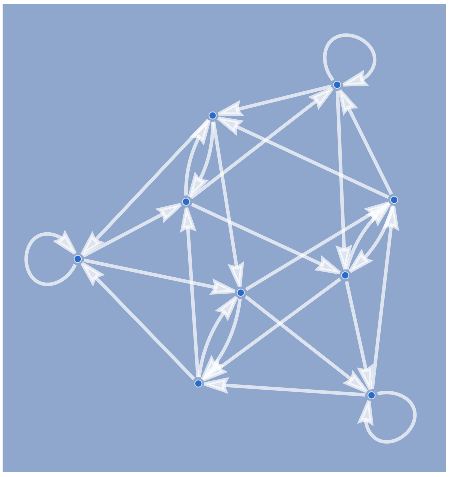
```
? IGDeBruijnGraph
```

IGDeBruijnGraph[m, n] gives a De Bruijn graph on m characters and string length n.

In[107]:=

```
IGDeBruijnGraph[3, 2, GraphStyle → "BackgroundBlue", EdgeStyle → Thick]
```

Out[107]=



## IGRealizeDegreeSequence

In[108]:=

```
? IGRealizeDegreeSequence
```

IGRealizeDegreeSequence[degrees] gives an undirected graph having the given degree sequence. Available Method options: {"SmallestFirst", "LargestFirst", "Index"}.
IGRealizeDegreeSequence[indegrees, outdegrees] gives a directed graph having the given out– and in–degree sequences.

This function constructs an undirected graph with the given degree sequence, or a directed graph with the given in- and out-degree sequences. For constructing simple graphs, it uses the Havel–Hakimi algorithm (undirected case) or Kleitman–Wang algorithm (directed case). These algorithms work by selecting a “hub” vertex, and connecting up all its free (out-)degrees to other vertices with the largest degrees. In the directed case, the “largest” degrees are determined by lexicographic ordering of (in, out)-degree pairs. For constructing a loop-free multigraph, a similar algorithm is used, but the hub is connected to only one other vertex in each step, instead of to as many as its degree. If self-loops are allowed as well, the same algorithm is used, and if a loop-free result cannot be created, an appropriate number of self-loops will be added to the very last hub. The order in which hub vertices are selected is controlled by the `Method` option.

To randomly sample multiple realizations of a degree sequence, use `IGDegreeSequenceGame`, or first create a single graph with `IGRealizeDegreeSequence`, then randomly rewire it using `IGRewire`.

The available options are:

- `SelfLoops → True` allows creating self-loops (disallowed by default).
- `MultiEdges → True` allows creating multi-edges (disallowed by default).
- The `Method` option controls the order in which hub vertices are chosen.

Available `Method` option values:

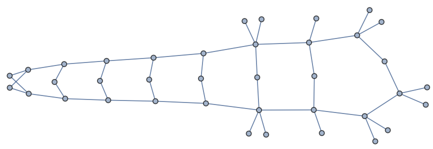
- `"SmallestFirst"` will choose a smallest-degree vertex in each step of the algorithm (this is the default). This results in a disassortative network. In the undirected case, this method is guaranteed to construct a connected graph, if one exists, both when constructing simple graphs or multigraphs. See Horvát and Modes (2020), as well as http://szhorvat.net/pelican/hh-connected-graphs.html for the proof. In the directed case, it tends to construct weakly-connected graphs, but this is not guaranteed.
- `"LargestFirst"` will choose a largest-degree vertex. This results in an assortative network. This method tends to construct disconnected graphs. This is the most common variant of the Havel–Hakimi algorithm implemented in other packages.

- "Index" will choose vertices in the order of their indices.

In[109]:=
```
degseq = VertexDegree@IGGiantComponent@RandomGraph[{50, 50}]
```
Out[109]=
```
{3, 4, 4, 4, 2, 1, 2, 3, 3, 1, 2, 3, 2, 3, 1, 5,
 2, 4, 3, 1, 2, 5, 4, 3, 1, 3, 2, 1, 2, 2, 3, 1, 3, 1, 1, 1, 1, 1}
```

The default Method → "SmallestFirst" tends to create highly disassortative graphs. The result is guaranteed to be connected if the input degree sequence was potentially connected.

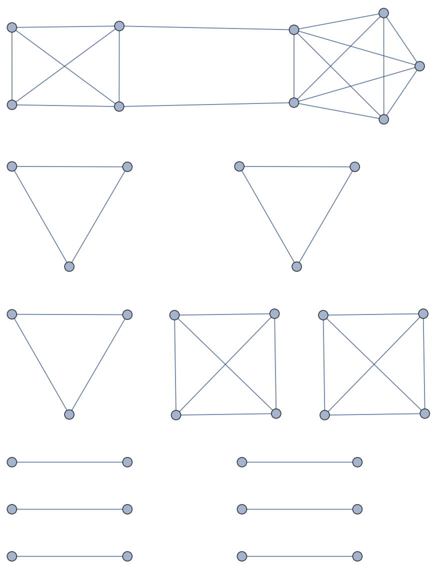
In[110]:=
```
IGRealizeDegreeSequence[degseq]
```
Out[110]=



In[111]:=
```
N@GraphAssortativity[%]
```
Out[111]=
```
-0.524347
```

Method → "LargestFirst" tends to create highly assortative disconnected graphs.

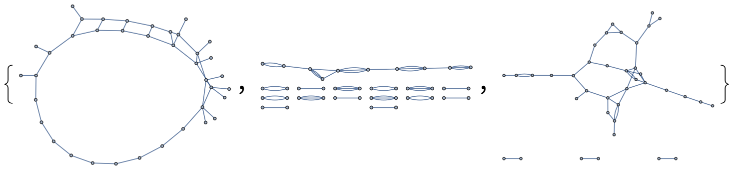
In[112]:=
```
IGRealizeDegreeSequence[degseq, Method → "LargestFirst"]
```
Out[112]=



In[113]:=
```
N@GraphAssortativity[%]
```
Out[113]=
```
0.904728
```

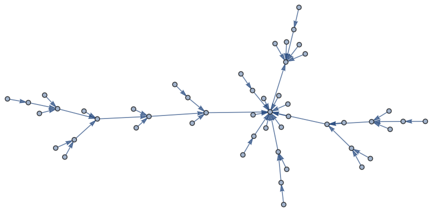
Allow parallel edges.

In[114]:=
```
IGRealizeDegreeSequence[degseq, MultiEdges → True, Method → #] & /@
 {"SmallestFirst", "LargestFirst", "Index"}
```
Out[114]=

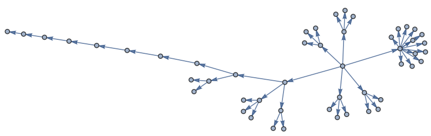


Create a directed graph.

In[115]:=
```
g = IGBarabasiAlbertGame[50, 1]
```
Out[115]=



In[116]:=
```
indegseq = VertexInDegree[g];
outdegseq = VertexOutDegree[g];
```
In[118]:=
```
IGRealizeDegreeSequence[outdegseq, indegseq]
```
Out[118]=



Verify that the degrees sequences of the result match the input to the function.

In[119]:=
```
{VertexOutDegree[%] == outdegseq, VertexInDegree[%] == indegseq}
```
Out[119]=
```
{False, False}
```

## References


- S. L. Hakimi, On Realizability of a Set of Integers as Degrees of the Vertices of a Linear Graph, Journal of the Society for Industrial and Applied Mathematics 10, 3 (1962). http://eudml.org/doc/19050
- V. Havel, Poznámka O Existenci Konečných Grafů (A Remark on the Existence of Finite Graphs), Časopis Pro Pěstování Matematiky 80, 4 (1955). https://www.jstor.org/stable/2098746
- D. J. Kleitman and D. L. Wang, Algorithms for Constructing Graphs and Digraphs with Given Valences and Factors, Discrete Mathematics 6, 1 (1973). https://doi.org/10.1016/0012-365X(73)90037-X
- Sz. Horvát and C. D. Modes, Connectivity matters: Construction and exact random sampling of connected graphs (2020). https://arxiv.org/abs/2009.03747

## IGGraphAtlas

In[120]:=

```
? IGGraphAtlas
```

IGGraphAtlas[n] gives graph number n from An Atlas of Graphs by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998. This function is provided for convenience; if you are looking for a specific named graph, use the builtin GraphData function.

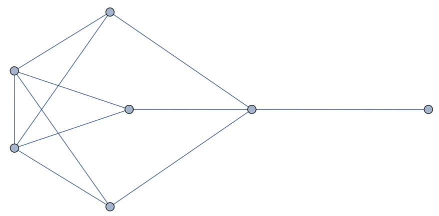
This function is provided for convenience for those who have the book *An Atlas of Graphs* by Ronald C. Read and Robin J. Wilson, and for those who wish to replicate results obtained with other packages that include this database. For all other purposes, use *Mathematica*'s built-in GraphData function.

Retrieve graph number 789:

In[121]:=

```
IGGraphAtlas[789]
```

Out[121]=



## IGFamousGraph

In[122]:=

```
? IGFamousGraph
```

IGFamousGraph[name] returns the given graph from igraph's built–in database.

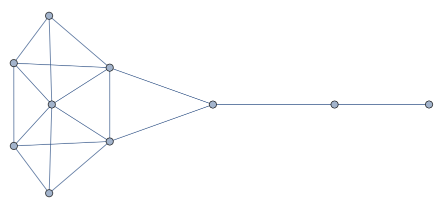
This function returns various “named” graphs from the igraph C library's built-in database. It is included in IGraph/M for compatibility with other igraph interfaces. It is recommended to use Mathematica's built-in `GraphData` function instead. See the documentation of the igraph C library for the list of supported graph names.

Create Krackhardt's kite graph:

In[123]:=

```
g = IGFamousGraph["Krackhardt_Kite"]
```

Out[123]=

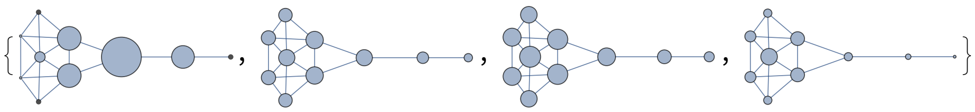


Krackhardt's kite was devised to illustrate the difference between various centrality measures:

In[124]:=

```
#[g] & /@ {
   IGVertexMap[0.1 # &, VertexSize → IGBetweenness],
   IGVertexMap[# &, VertexSize → IGCloseness],
   IGVertexMap[# &, VertexSize → IGHarmonicCentrality],
   IGVertexMap[0.1 # &, VertexSize → VertexDegree]
  }
```

Out[124]=



## IGFromNauty

In[125]:=

```
? IGFromNauty
```

IGFromNauty[string] interprets a Graph6, Digraph6 or Sparse6 string representation of a graph.

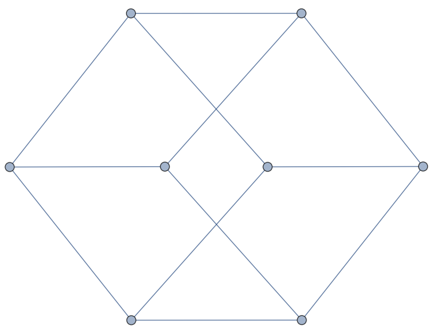
`IGFromNauty` converts a Graph6, Digraph6 or Sparse6 string to a graph. These formats originate with the nauty suite of programs and are supported by many other graph theory software.

Interpret a Graph6 string.

In[126]:=

```
IGFromNauty["Gr`HOk"]
```

Out[126]=

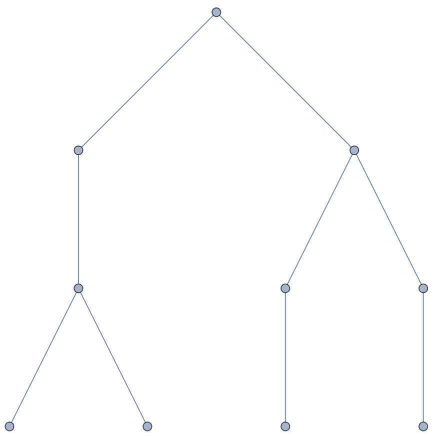


Interpret a Sparse6 string. These start with a `:` character.

In[127]:=

```
IGFromNauty[":I`ESgTlVF"]
```

Out[127]=

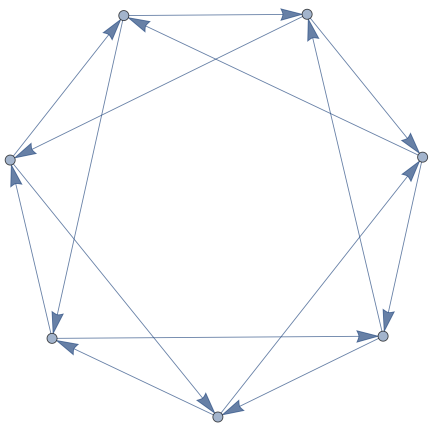


Interpret a Digraph6 string. These start with a & character.

In[128]:=

```
IGFromNauty["&FKB?oMB_W?"]
```

Out[128]=



`IGFromNauty` does not support headers or whitespace in the string. To handle these, or to interpret a multiline string, use `IGImportString[…, "Nauty"]`.

In[129]:=

```
IGFromNauty[">>graph6<<DYw"]
```

IGraphM: Illegal character in Graph6, Digraph6 or Sparse6 line.

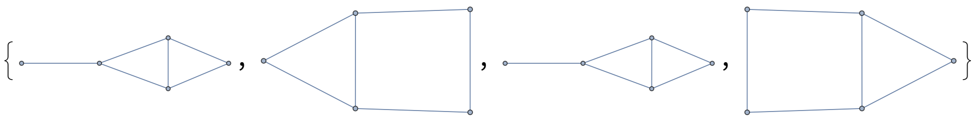
Out[129]=

```
$Failed
```

In[130]:=

```
IGImportString[
 ">>graph6<<DYw
Dhs
Dxo
DVW
", "Nauty"]
```

Out[130]=



## Random graph generators

These graph creation functions use igraph’s random graph generator, which can be seeded using `IGSeedRandom`.

In[131]:=

```
? IG*Game*
```

▼ IGraphM`

| | | |
|---|---|---|
| IGAsymmetricPreferenceGame | IGErdosRenyiGameGNP | IGStaticFitnessGame |
| IGBarabasiAlbertGame | IGEstablishmentGame | IGStaticPowerLawGame |
| IGBipartiteGameGNM | IGForestFireGame | IGStochasticBlockModelGame |
| IGBipartiteGameGNP | IGGeometricGame | IGTreeGame |
| IGCallawayTraitsGame | IGGrowingGame | IGWattsStrogatzGame |
| IGDegreeSequenceGame | IGKRegularGame | |
| IGErdosRenyiGameGNM | IGPreferenceGame | |

### Basic random graphs

In[132]:=

```
? IGErdosRenyiGameGNM
```

IGErdosRenyiGameGNM[n, m] generates a random graph with n vertices and m edges.

In[133]:=

```
? IGErdosRenyiGameGNP
```

IGErdosRenyiGameGNP[n, p] generates a random graph on n vertices, in which each edge is present with probability p.

`IGErdosRenyiGameGNM` uniformly samples graphs with $n$ vertices and $m$ edges. This random graph model is known as the Erdős–Rényi $G(n, m)$ model.

In `IGErdosRenyiGameGNP`, each edge is present with the same and independent probability. This model is known as the Erdős–Rényi $G(n, p)$ model or Gilbert model.

The available options are:

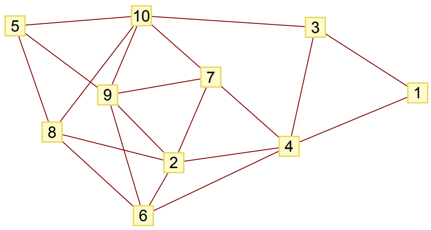
- `DirectedEdges` → `True` produces a directed graph.
- `SelfLoops` → `True` allows self-loops.

Create a random graph with 10 vertices and 20 edges.

In[134]:=

```
IGErdosRenyiGameGNM[10, 20, GraphStyle → "VintageDiagram"]
```

Out[134]=

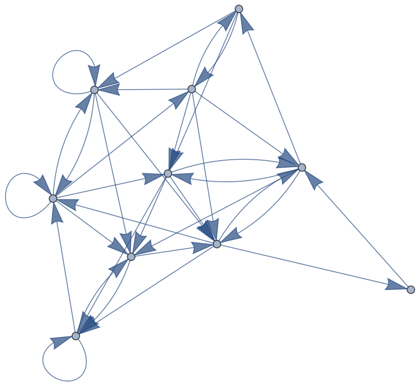



Create a directed graph and allow self-loops.

In[135]:=

```
IGErdosRenyiGameGNM[10, 35, DirectedEdges → True, SelfLoops → True]
```

Out[135]=

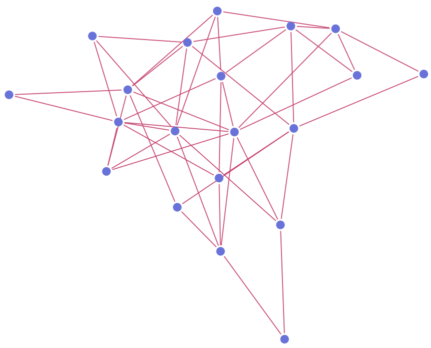


Insert each edge with a probability of 20%.

In[136]:=

```
IGErdosRenyiGameGNP[20, 0.2, GraphStyle → "RoyalColor"]
```

Out[136]=

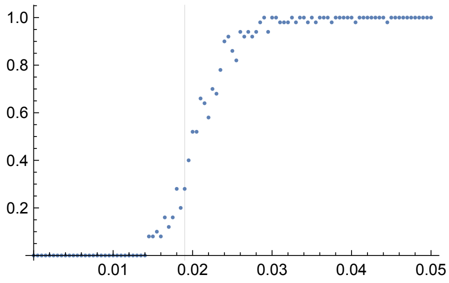


The $G(n, p)$ model produces connected graphs with high probability for $p > \ln(n)/n$.

In[137]:=

```
n = 300;
ListPlot[
 Table[
  {p, Mean@Boole@Table[ConnectedGraphQ@IGErdosRenyiGameGNP[n, p], {50}]},
  {p, 0, 0.05, 0.0005}
 ],
 GridLines → {{Log[n] / n}, None}
]
```

Out[138]=




## Random bipartite graphs

In[139]:=

```
? IGBipartiteGameGNM
```

IGBipartiteGameGNM[n1, n2, m] generates a bipartite random graph with n1 and n2 vertices in the two partitions and m edges.

In[140]:=

```
? IGBipartiteGameGNP
```

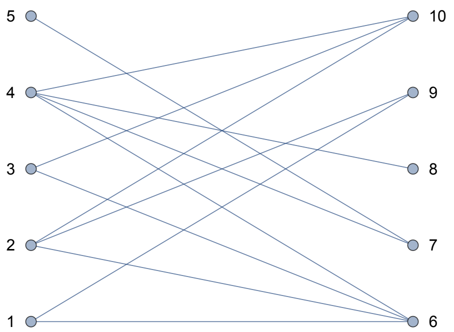
IGBipartiteGameGNP[n1, n2, p] generates a bipartite Bernoulli random graph with n1 and n2 vertices in the two partitions and connection probability p.

`IGBipartiteGameGNM` and `IGBipartiteGameGNP` are equivalent to `IGErdosRenyiGNM` and `IGErdosRenyiGNP`, but they generate bipartite graphs.

The available options are:

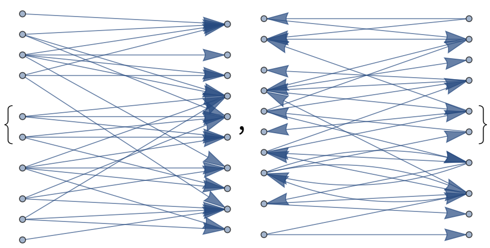
- `DirectedEdges → True` creates a directed graph.
- `"Bidirectional" → True` allows directed edges to run in either direction between the two partitions. The default is `False`, which means that edges will run only from the first partition to the second. This option is ignored for undirected graphs.

In[141]:=
```
IGBipartiteGameGNP[5, 5, 0.5, VertexLabels → "Name"]
```
Out[141]=




Create a bipartite directed graph with edges running either uni-directionally or bidirectionally between the two partitions.

In[142]:=
```
IGLayoutBipartite@IGBipartiteGameGNM[10, 10, 30, DirectedEdges → True, "Bidirectional" → #] & /@
 {False, True}
```
Out[142]=



## IGTreeGame

In[143]:=
```
? IGTreeGame
```

IGTreeGame[n] generates a random tree on n vertices. Sampling is uniform over
the set of labelled trees. Available Method options: {"PruferCode", "LoopErasedRandomWalk"}.

`IGTreeGame` samples uniformly from the set of labelled trees.

Available options:

- `DirectedEdges → True` will create a directed tree, with edges oriented away from the root.
- `Method` can be used to choose the tree generation algorithm. All methods sample labelled trees uniformly.

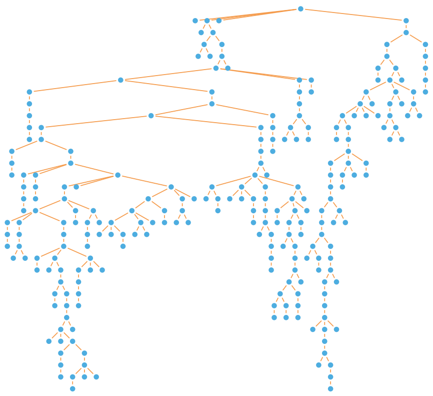
Available `Method` options:

- `"PruferCode"`, works by generating a random Prüfer sequence, then constructing a tree from it. It does not currently support directed trees.
- `"LoopErasedRandomWalk"`, uses a loop-erased random walk to uniformly sample the spanning trees of the complete graph.

In[144]:=

```
IGTreeGame[250, GraphLayout → "LayeredEmbedding", PlotTheme → "PastelColor"]
```

Out[144]=

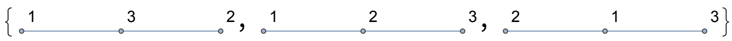


There are several distinct labellings of isomorphic trees. All of these are generated with equal probability.

In[145]:=

```
Table[
  IGTreeGame[3, VertexLabels → Automatic],
  {100}
 ] // DeleteDuplicatesBy[AdjacencyMatrix]
```

Out[145]=

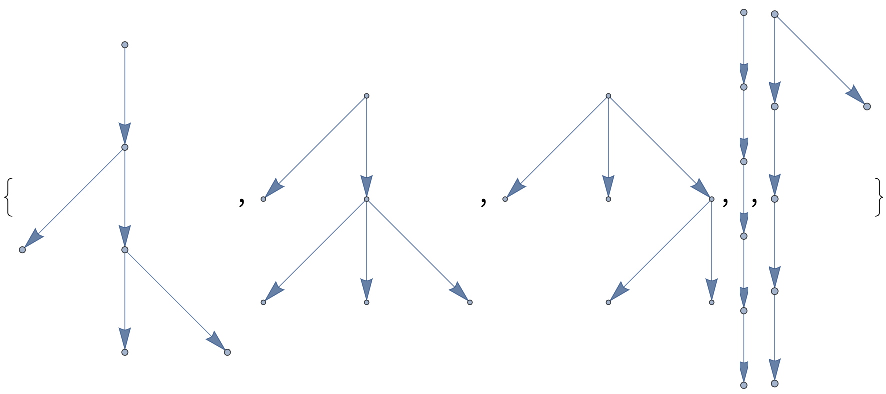



Generate directed trees.

In[146]:=

```
Table[IGTreeGame[6, DirectedEdges → True, GraphLayout → "LayeredDigraphEmbedding"], {5}]
```

Out[146]=



Generate a random sparse connected graph by first creating a tree, then adding cycle edges. Note that this method does not sample connected graphs uniformly.

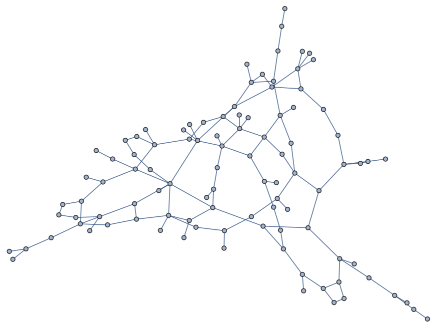
In[147]:=

```
randomConnected[nodeCount_, edgeCount_] :=
 Module[{tree},
  tree = IGTreeGame[nodeCount];
  EdgeAdd[tree, RandomSample[EdgeList@GraphComplement[tree], edgeCount - nodeCount + 1]]
 ]
```

In[148]:=

```
randomConnected[100, 120]
```

Out[148]=

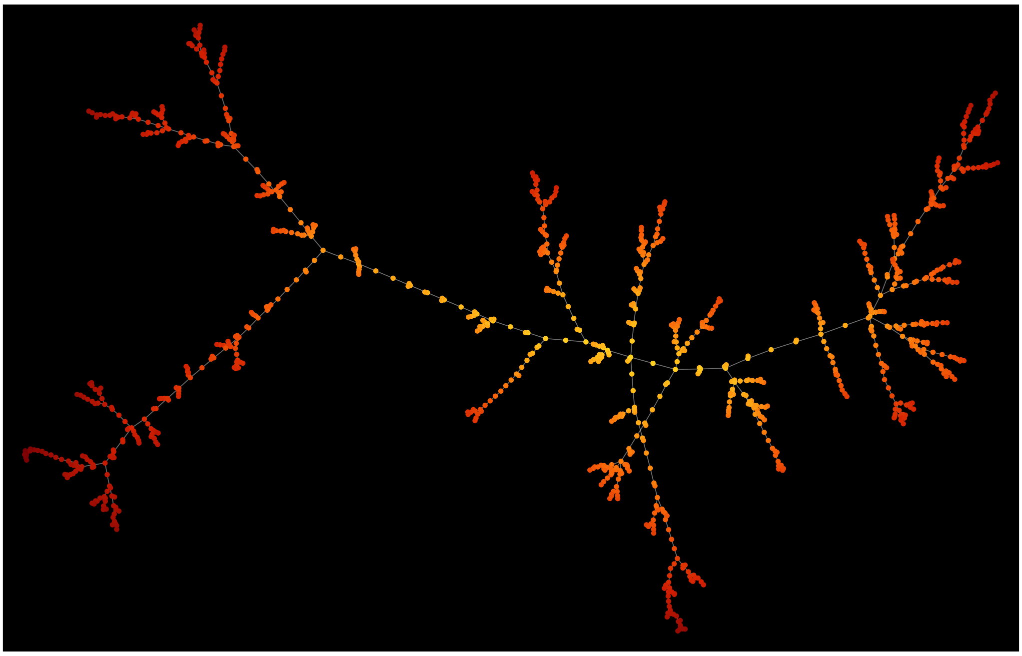


Colour the nodes of a random tree by their inverse average distance to other nodes.

In[149]:=

```
IGVertexMap[
 ColorData["SolarColors"],
 VertexStyle → Rescale@*IGCloseness,
 IGTreeGame[1000, Background → Black, ImageSize → Large, EdgeStyle → LightGray]
]
```

Out[149]=



## IGDegreeSequenceGame

In[150]:=

```
? IGDegreeSequenceGame
```

IGDegreeSequenceGame[degrees] generates an undirected random graph with the given degree sequence. Available Method options: {"ConfigurationModel", "ConfigurationModelSimple", "FastSimple", "VigerLatapy"}.
IGDegreeSequenceGame[indegrees, outdegrees]
generates a directed random graph with the given in– and out–degree sequences.

`IGDegreeSequenceGame` implements various random sampling methods for graphs with a given degree sequence. To quickly construct a single realization of a degree sequence, use `IGRealizeDegreeSequence`.

`IGDegreeSequenceGame` takes the following values for its `Method` option:

- `"ConfigurationModel"` implements the configuration model: it connects up vertex stubs randomly. It may generate both self-loops and multi-edges. Undirected graphs are generated with probability proportional to $\left(\prod_{i<j} A_{ij}! \prod_i A_{ii}!!\right)^{-1}$, where $A$ is the adjacency matrix, having *twice* the number of loops for each vertex on the diagonal. Directed ones are generated with probability proportional to $\left(\prod_{i,j} A_{ij}!\right)^{-1}$.

  All simple graphs are generated with the same probability, but the probability of multigraphs and graphs with self-loops differs from that of simple graphs and depends on their specific structure.
- `"ConfigurationModelSimple"` also implements the configuration model, but it rejects non-simple graphs. It samples uniformly from the set of all simple graphs with the given degree sequence. This method can be very slow for dense graphs.
- `"FastSimple"` is a fast generation algorithm that avoids self-loops and multi-edges. This method can generate any simple graph with the given degree sequence, but it does not sample them uniformly.
- `"VigerLatapy"` can sample undirected, connected simple graphs uniformly and uses Monte-Carlo methods to randomize the graphs. This generator should be favoured if undirected and connected graphs are to be generated and execution time is not a concern. igraph uses the original implementation of Fabien Viger; see https://www-complexnetworks.lip6.fr/~latapy/FV/generation.html and the corresponding paper at https://arxiv.org/abs/cs/0502085.

The default method is `"FastSimple"`. Note that it does not sample uniformly.

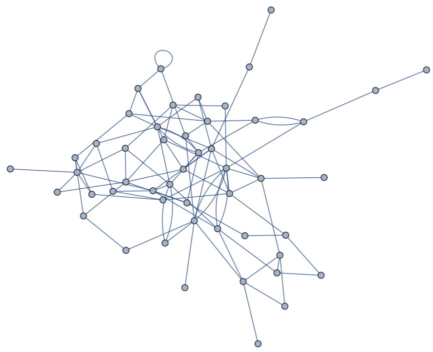
In[151]:=

```
degseq = VertexDegree@RandomGraph[{50, 100}];
```

In[152]:=

```
IGDegreeSequenceGame[degseq, Method → "ConfigurationModel"]
```

Out[152]=



In[153]:=

```
SimpleGraphQ[%]
```

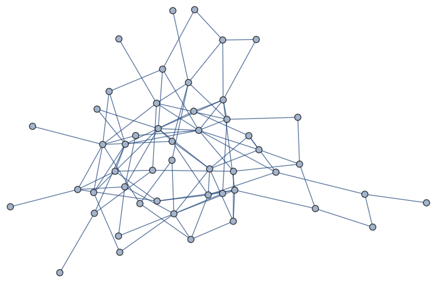
Out[153]=

```
False
```

In[154]:=

```
IGDegreeSequenceGame[degseq, Method → "ConfigurationModelSimple"]
```

Out[154]=



In[155]:=

```
SimpleGraphQ[%]
```

Out[155]=

```
True
```

The configuration model algorithm is too slow to construct even small dense graphs.

In[156]:=

```
ds = VertexDegree@RandomGraph[{10, Binomial[10, 2] - 5}]
```

Out[156]=

```
{9, 7, 7, 9, 9, 8, 8, 7, 8, 8}
```

In[157]:=

```
TimeConstrained[IGDegreeSequenceGame[ds, Method → "ConfigurationModelSimple"], 1]
```

Out[157]=

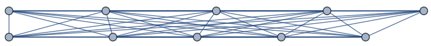
```
$Aborted
```

Graphs that are almost complete can be sampled by generating the complement first.

In[159]:=

```
GraphComplement@IGDegreeSequenceGame[9 - ds, Method → "ConfigurationModelSimple"]
```

Out[159]=



In[160]:=

```
ds == VertexDegree[%]
```

Out[160]=

```
True
```

## IGKRegularGame

In[161]:=

```
? IGKRegularGame
```

IGKRegularGame[n, k] generates a k–regular graph on n vertices, i.e. a graph in which all vertices have degree k.

In a *k*-regular graph all vertices have degree *k*. The current implementation is able to generate any *k*-regular graph, but it does not sample them with precisely the same probability.

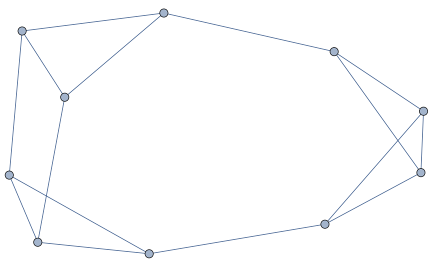
The available options are:

- `DirectedEdges → True` creates a directed graph.
- `MultiEdges → True` allows the creation of parallel edges.

In[162]:=

```
IGKRegularGame[10, 3]
```

Out[162]=



Not all parameters are valid:

In[163]:=
```
IGKRegularGame[5, 3]
```

IGraphM: src/games/degree_sequence.c:151 – No simple undirected graph can realize the given degree sequence.

IGraphM: igraph returned with error: Invalid value.

Out[163]=
```
$Failed
```

There are no graphs with 5 vertices each having degree 3.

In[164]:=
```
IGGraphicalQ[{3, 3, 3, 3, 3}]
```

Out[164]=
```
False
```

## IGGrowingGame

In[165]:=
```
? IGGrowingGame
```

IGGrowingGame[n, k] generates a growing random graph with n vertices, adding a new vertex and k new edges in each step.

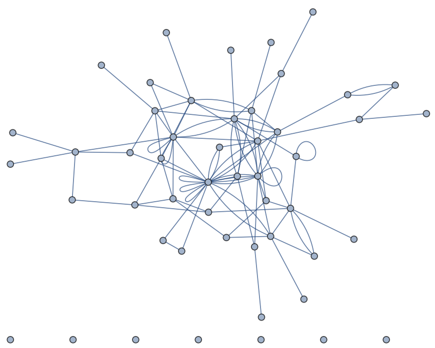
`IGGrowingGame[n, k]` creates a random graph by successively adding vertices to the graph until the vertex count n is reached. At each step, k new edges are added as well.

The available options are:

- `DirectedEdges → True` creates a directed graph.
- `"Citation" → True` connects newly added edges to the newly added vertex.

In[166]:=
```
IGGrowingGame[50, 2]
```

Out[166]=

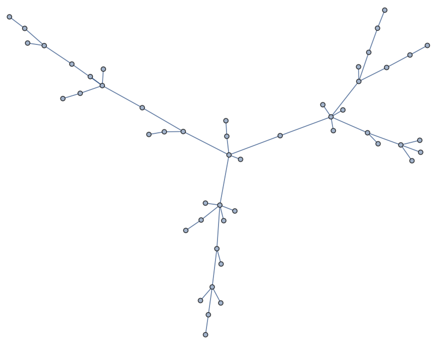


With "Citation" → True, the newly added edges are connected to the newly added vertices.

In[167]:=

```
IGGrowingGame[50, 1, "Citation" → True]
```

Out[167]=



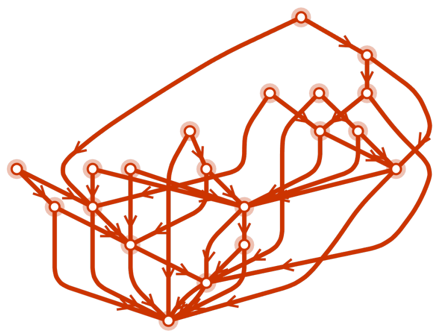
Note that while this model can be used to generate random trees, it will not sample them uniformly. If uniform sampling is desired, use IGTreeGame instead.

Create a directed citation graph.

In[168]:=

```
IGGrowingGame[20, 2, DirectedEdges → True, "Citation" → True, GraphStyle → "Web"]
```

Out[168]=



## IGBarabasiAlbertGame

In[169]:=

```
? IGBarabasiAlbertGame
```

IGBarabasiAlbertGame[n, k] generates an n–vertex Barabási–Albert random graph by adding a new vertex with k (out–)edges in each step. Available Method options: {"Bag", "PSumTree", "PSumTreeMultiple"}.
IGBarabasiAlbertGame[n, {k2, k3, …}] generates an n–vertex Barabási–Albert random graph by adding a new vertex with k2, k3, … out–edges in each step.
IGBarabasiAlbertGame[n, k, {β, A}] generates a Barabási–Albert random graph with preferential attachment probabilities proportional to d^β + A where d is the vertex (in–)degree.

IGBarabasiAlbertGame implements a preferential attachment model. It generates a graph by sequentially adding new vertices with the specified number of edges ($k$). The edges will connect to existing vertices with probability $d^{\beta} + A$, where $d$ is the in-degree of the existing vertex. The default parameters are $\beta = 1$ and $A = 1$.

The available options are:

- DirectedEdges → False creates an undirected graph.

- "TotalDegreeAttraction" → True computes the attachment probability based on the the total degree of existing vertices (i.e. the sum of in- and out-degrees), not their in-degree. Always assumed to be True when using DirectedEdges → True.
- "StartingGraph" → g will use graph g as the starting point for building the preferential attachment graph. The vertex names of g are ignored; the result always uses positive integers as vertex names.

Available Method option values:

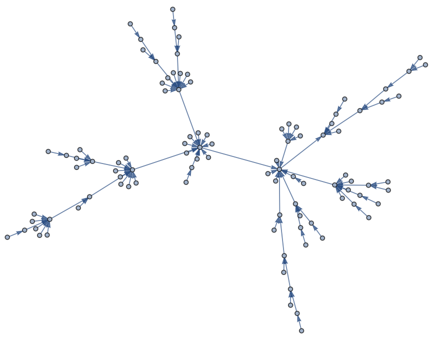
- "Bag" works by putting the IDs of the vertices into a bag exactly as many times as their (in-)degree, plus once more. Then the required number of cited vertices are drawn from the bag, with replacement. This method might generate multi-edges. It only works if $\beta = 1$ and $A = 1$.
- "PSumTree" uses a partial prefix-sum tree to generate the graph. It does not generate multi-edges and works for any $\beta$ and $A$ values.
- "PSumTreeMultiple" works like "PSumTree" but allows multi-edges.

The built-in BarabasiAlbertGraphDistribution is equivalent to using $A = 0$ and DirectedEdges → False in IGBarabasiAlbertGame, while the built-in PriceGraphDistribution is equivalent DirectedEdges → True.

In[170]:=

```
IGBarabasiAlbertGame[100, 1]
```

Out[170]=

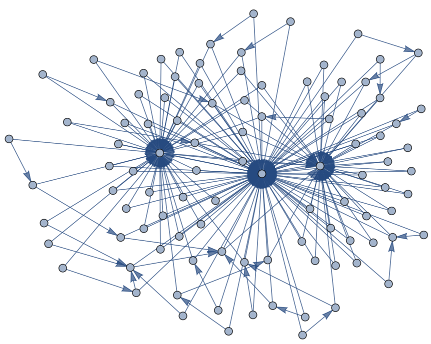


Use attachment probability proportional to $\texttt{degree}^{1.5} + 1$.

In[171]:=

```
IGBarabasiAlbertGame[100, 2, {1.5, 1}]
```

Out[171]=

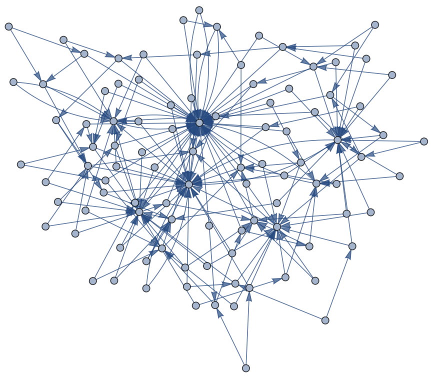


The "Bag" method may generate parallel edges:

In[172]:=

```
IGBarabasiAlbertGame[100, 2, Method → "Bag"]
```

Out[172]=



In[173]:=

```
MultigraphQ[%]
```

Out[173]=

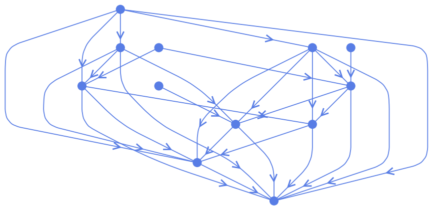
```
True
```

Create a graph with the given out-degree sequence. The $k^{th}$ entry in the degree sequence list must be no greater than $k$.

In[174]:=

```
IGBarabasiAlbertGame[12, {1, 2, 3, 2, 1, 3, 4, 5, 1, 5, 2}, PlotTheme → "Minimal"]
```

Out[174]=



In[175]:=

```
VertexOutDegree[%]
```

Out[175]=

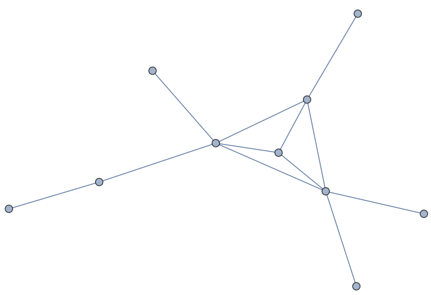
```
{0, 1, 2, 3, 2, 1, 3, 4, 5, 1, 5, 2}
```

Create a preferential attachment graph using a 4-node complete graph as the starting point.

In[176]:=

```
IGBarabasiAlbertGame[10, 1, "StartingGraph" → CompleteGraph[4]]
```

Out[176]=



## IGWattsStrogatzGame

In[177]:=

```
? IGWattsStrogatzGame
```

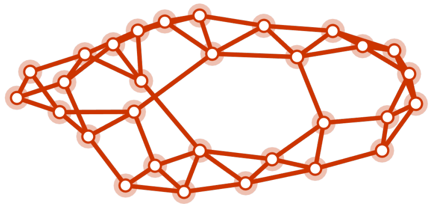
IGWattsStrogatzGame[n, p] generates an n–vertex Watts–Strogatz random graph using rewiring probability p.
IGWattsStrogatzGame[n, p, k] rewires a lattice where each vertex is connected to its k–neighbourhood.
IGWattsStrogatzGame[n, p, {dim, k}] rewires a dim dimensional
lattice of n^dim vertices, where each vertex is connected to its k–neighbourhood.

The two-argument form produces results equivalent to that of the built-in `WattsStrogatzGraphDistribution`.

In[178]:=

```
IGWattsStrogatzGame[30, 0.05, PlotTheme → "Web"]
```

Out[178]=

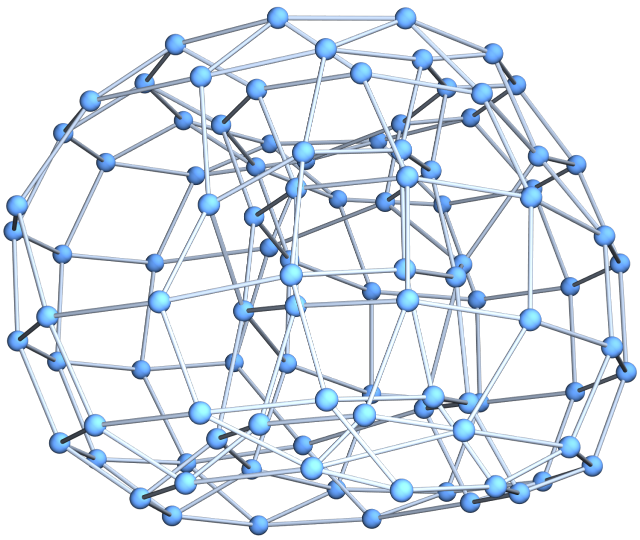


The extended form allows for multi-dimensional lattices. Create a graph by randomly rewiring a two-dimensional toroidal lattice of 10 × 10 nodes:

In[179]:=

```
Graph3D@IGWattsStrogatzGame[10, 0.01, {2, 1}]
```

Out[179]=



## IGStaticFitnessGame

In[180]:=

```
? IGStaticFitnessGame
```

IGStaticFitnessGame[m, {f1, f2, …}] generates a random undirected
graph with m edges where edge i <–> j is inserted with probability proportional to f_i×f_j.
IGStaticFitnessGame[m, {fout1, fout2, …}, {fin1, fin2, …}] generates a random directed
graph with m edges where edge i –> j is inserted with probability proportional to fout_i×fin_j.

`IGStaticFitnessGame` generates a random graph by connecting vertices based on their fitness score. The algorithm starts with *n* vertices and no edges. Two vertices are selected with probabilities proportional to their fitness scores (for directed graphs, a starting vertex is selected based on its out-fitness and an end vertex based on its in-fitness). If they are not yet connected, an edge is inserted between them. The procedure is repeated until the number of edges reaches $m$.

The expected degree of each vertex is proportional to its fitness score. This is exactly true when self-loops and multi-edges are allowed, and approximately true otherwise.

`IGStaticFitnessGame` approximates the Chung–Lu model in which each edge `i` ⟷ `j` is present independently, with probability

$$p_{ij} = \begin{cases} \frac{f_i f_j}{2m} & \text{if } i \neq j \\ \frac{f_i f_j}{4m} & \text{if } i = j \end{cases},$$

where $m = \frac{1}{2} \sum_k f_k$.

Unlike the Chung–Lu algorithm, which would require $O(m^2)$ computation steps, `IGStaticFitnessGame` runs in $O(m)$ time.

The available options are:

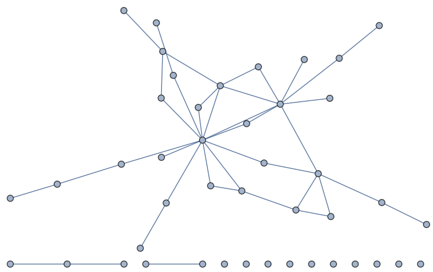
- `SelfLoops → True` allows the creation of self-loops.
- `MultiEdges → True` allows the creation of parallel edges.

Create an undirected graph with four high-degree nodes and 40 low-degree ones.

In[181]:=

```
weights = Join[{10, 10, 10, 10}, ConstantArray[1, 40]];
IGStaticFitnessGame[Total[weights] / 2, weights]
```

Out[182]=



In[183]:=

```
VertexDegree[%]
```

Out[183]=

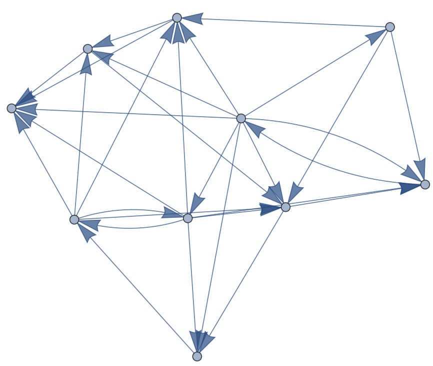
```
{5, 5, 12, 8, 2, 2, 1, 2, 2, 1, 1, 2, 2, 0, 2, 1, 2, 0, 0,
 3, 1, 1, 0, 0, 1, 2, 0, 3, 0, 2, 1, 2, 1, 1, 1, 2, 0, 1, 2, 0, 0, 3, 2, 1}
```

Create a directed graph.

In[184]:=

```
IGStaticFitnessGame[30, Range[10], Range[10, 1, -1]]
```

Out[184]=



When self-loops and multi-edges are allowed, the expected degree of each vertex is proportional to its fitness score.

In[185]:=

```
degrees = {3, 3, 2, 2, 2, 1, 1};
Table[
   VertexDegree@IGStaticFitnessGame[
     Total[degrees] / 2, degrees,
     SelfLoops → True, MultiEdges → True
    ],
   {1000}
  ] // N // Mean
```

Out[186]=

```
{3.03, 2.97, 2.023, 1.957, 2.056, 0.977, 0.987}
```

When generating simple graphs, this holds only approximately.

In[187]:=

```
degrees = {3, 3, 2, 2, 2, 1, 1};
Table[
   VertexDegree@IGStaticFitnessGame[
     Total[degrees] / 2, degrees
    ],
   {1000}
  ] // N // Mean
```

Out[188]=

```
{2.703, 2.625, 2.04, 2.071, 2.086, 1.23, 1.245}
```

## IGStaticPowerLawGame

In[189]:=

```
? IGStaticPowerLawGame
```

IGStaticPowerLawGame[n, m, exp] generates a random graph with n vertices and m edges, having a power–law degree distribution with the given exponent.
IGStaticPowerLawGame[n, m, expOut, expIn] generates a random directed graph with n vertices and m edges, having power–law in– and out–degree distributions with the given exponents.

`IGStaticPowerLawGame` generates a directed or undirected random graph where the degrees of vertices follow power-law distributions with prescribed exponents. For directed graphs, the exponents of the in- and out-degree distributions may be specified separately.

This function is equivalent to `IGStaticFitnessGame` with a fitness vector $f$ where $f_i = i^{-\alpha}$ and $\alpha = \frac{1}{\text{exponent}-1}$.

Note that significant finite size effects may be observed for exponents smaller than 3 in the original formulation of the game. This function removes the finite size effects by default by assuming that the fitness of vertex $i$ is $(i + i_0)^{-\alpha}$, where $i_0$ is a constant chosen appropriately to ensure that the maximum degree is less than the square root of the number of edges times the average degree; see the paper of Chung and Lu, and Cho et al. for more details.

The available options are:

- `SelfLoops → True` allows the creation of self-loops.
- `MultiEdges → True` allows the creation of parallel edges.
- `"FiniteSizeCorrection" → False` disables finite size correction, which is used by default.

Create a graph with a power-law degree distribution of exponent 2.5.

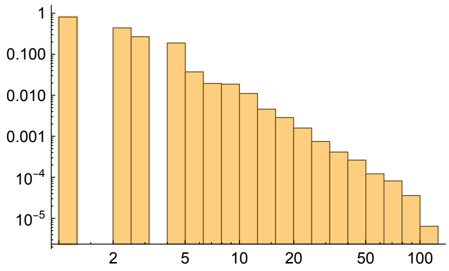
In[190]:=

```
g = IGStaticPowerLawGame[100 000, 200 000, 2.5];
```

In[191]:=

```
Histogram[VertexDegree[g], "Log", {"Log", "PDF"}]
```

Out[191]=

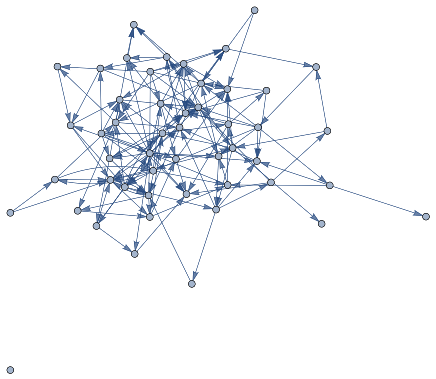



Create a directed graph with power-law in- and out-degree distributions.

In[192]:=

```
IGStaticPowerLawGame[50, 150, 3, 3]
```

Out[192]=



### References


- Goh K-I, Kahng B, Kim D: Universal behaviour of load distribution in scale-free networks. Phys Rev Lett 87(27):278701, 2001.
- Chung F and Lu L: Connected components in a random graph with given degree sequences. Annals of Combinatorics 6, 125-145, 2002.
- Cho YS, Kim JS, Park J, Kahng B, Kim D: Percolation transitions in scale-free networks under the Achlioptas process. Phys. Rev. Lett. 103:135702, 2009.


## IGStochasticBlockModelGame

In[193]:=

```
? IGStochasticBlockModelGame
```

IGStochasticBlockModelGame[ratesMatrix, blockSizes] samples from a stochastic block model.

The `ratesMatrix` argument gives the connection probability between and within blocks (groups of vertices). The `blockSizes` argument gives the size of each block (vertex group).

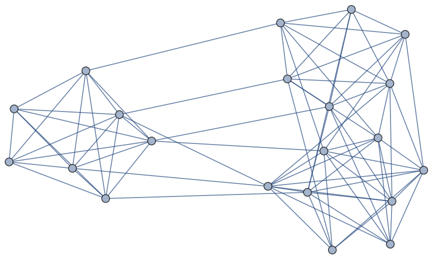
The available options are:

- `DirectedEdges` → `True` creates a directed graph.
- `SelfLoops` → `True` allows the creation of self-loops.

In[194]:=

```
IGStochasticBlockModelGame[(0.9  0.1  0.2
                            0.1  1    0.05
                            0.2  0.05 0.9), {6, 7, 8}]
```

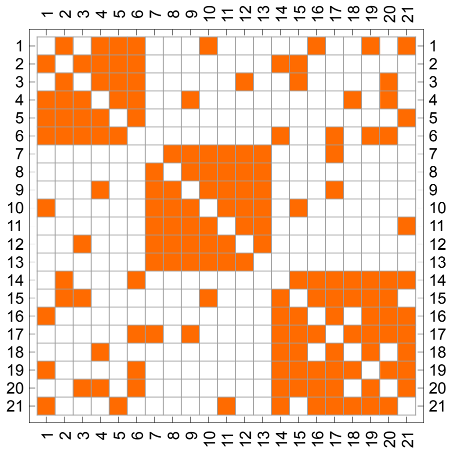
Out[194]=



In[195]:=

```
IGAdjacencyMatrixPlot[%]
```

Out[195]=



## IGPreferenceGame

In[196]:=

```
? IGPreferenceGame
```

IGPreferenceGame[n, typeWeights, preferenceMatrix]

**Experimental:** This is experimental functionality that may change without notice.

`IGPreferenceGame[n, w, p]` first samples n vertices of different types, each having type *i* with probability proportional to $w_i$. Then it connects vertices with types *i* and *j* with probability $p_{ij}$. This is similar to a stochastic block model, but the vertex types are chosen randomly.

The available options are:

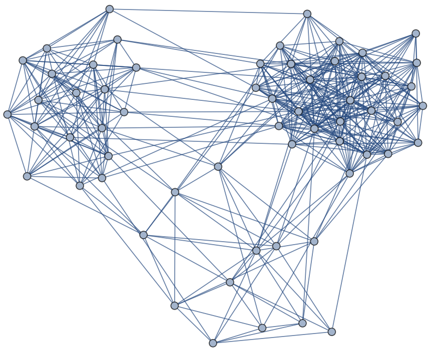
- `DirectedEdges → True` creates a directed graph.
- `SelfLoops → True` allows self-loops.

Generate a graph with three groups of vertices of different sizes, with intra-group connections being much more frequent than inter-group ones.

In[197]:=

```
IGPreferenceGame[60, {3, 6, 10}, 0.05 + 0.5 IdentityMatrix[3]]
```

Out[197]=

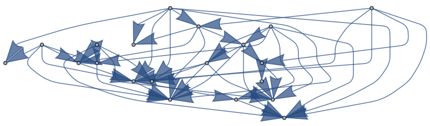


Generate a directed graph with low-probability intra-group connections and high probability unidirectional inter-group connections.

In[198]:=

```
IGPreferenceGame[20, {0.3, 0.7}, (0.05 0.3
                                  0    0.05), DirectedEdges → True]
```

Out[198]=



## IGAsymmetricPreferenceGame

In[199]:=

```
? IGAsymmetricPreferenceGame
```

IGAsymmetricPreferenceGame[n, typeWeightsMatrix, preferenceMatrix]

**Experimental:** This is experimental functionality that may change without notice.

`IGAsymmetricPreferenceGame[n, w, p]` is similar to `IGPreferenceGame[n, w, p]`, but it assigns a separate out-type and in-types to each vertex. The probability of a vertex having out-type *i* and in-type *j* is proportional to $w_{ij}$. The probability of connecting a vertex with out-type *i* to another one with in-type *j* is $p_{ij}$.

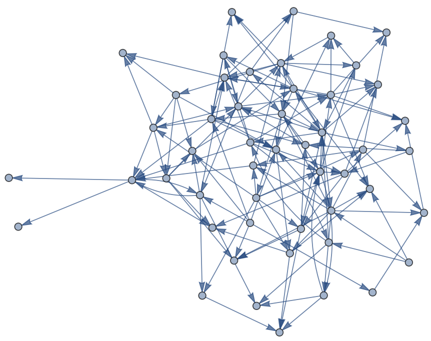
The available options are:

- `SelfLoops → True` allows self-loops.

In[200]:=

```
IGAsymmetricPreferenceGame[50, (0 1
                                0 2), (0.1  0.01
                                       0.01 0.1)]
```

Out[200]=



## IGForestFireGame

In[201]:=

```
? IGForestFireGame
```

IGForestFireGame[n, pForward] generates a graph on n vertices from the forest fire model.
IGForestFireGame[n, pForward, rBackward] specifies the backward to forward burning probability ratio (default: 1).
IGForestFireGame[n, pForward, rBackward, nAmbassadors]
also specifies the number of ambassador nodes in each step (default: 1).

The forest fire model is a growing graph model. In every time step, a new vertex is added to the graph. The new vertex chooses the specified number of ambassadors (default: 1) and starts a simulated forest fire at their locations. The fire spreads through the directed edges. The spreading probability along an edge is given by `pForward`. The fire may also spread backwards on an edge with probability `pForward * rBackward`. When the fire has ended, the newly added vertex connects to all the vertices that were burned in the fire.

The forest fire model intends to reproduce the following network characteristics, observed in real networks:

- Heavy-tailed in-degree and out-degree distributions.
- Community structure.
- Densification power-law. The network is densifying in time, according to a power-law rule.
- Shrinking diameter. The diameter of the network decreases in time.

The available options are:

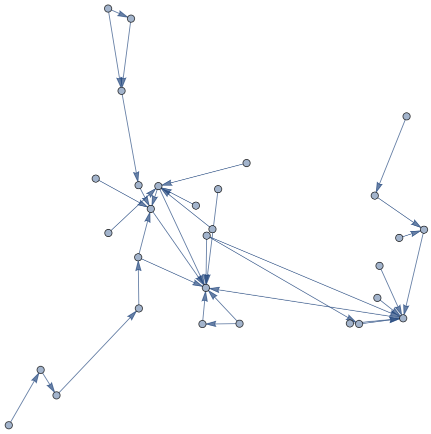
- `DirectedEdges → False` generates an undirected graph.

Generate a graph with only forward burning.

In[202]:=

```
IGForestFireGame[30, 0.2, 0,
 GraphLayout → "SpringEmbedding"]
```

Out[202]=

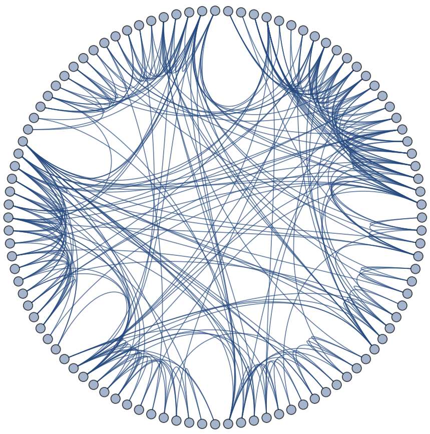


Generate a graph from the forest fire model, and visualize its community structure.

In[203]:=

```
IGForestFireGame[100, 0.2, 1, 2, DirectedEdges → False,
 GraphLayout → {"EdgeLayout" → "HierarchicalEdgeBundling"}]
```

Out[203]=

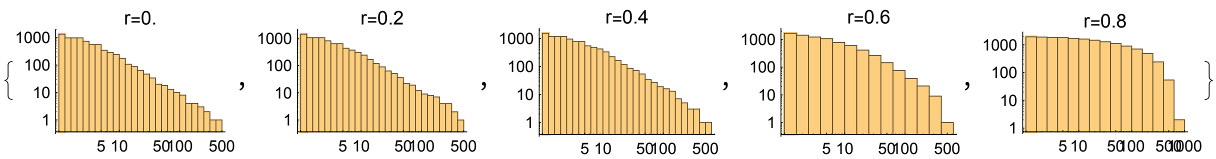


Plot the cumulative in-degree distribution for different backward to forward burning probability ratios.

In[204]:=

```
Table[
 Histogram[
  VertexInDegree@IGForestFireGame[2000, 0.4, r, 2, DirectedEdges → True],
  "Log", {"Log", "SurvivalCount"},
  PlotLabel → Row[{"r=", r}]
 ],
 {r, 0, 0.8, 0.2}
]
```

Out[204]=




### References


- Jure Leskovec, Jon Kleinberg and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2007. https://doi.org/10.1145/1217299.1217301
- Jure Leskovec, Jon Kleinberg and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 177–187, 2005.


## IGCallawayTraitsGame

In[205]:=

```
? IGCallawayTraitsGame
```

IGCallawayTraitsGame[n, k, typeWeights, preferenceMatrix]

This function simulates a growing random graph according to the following algorithm:

At each time step, a new vertex is added. Its type is randomly selected according to the type weights. Then k existing pairs of vertices are selected randomly, and each pair attempts to connect. The probability of success for given types of vertices is given by the preference matrix.

This algorithm may create self-loops and multi-edges.

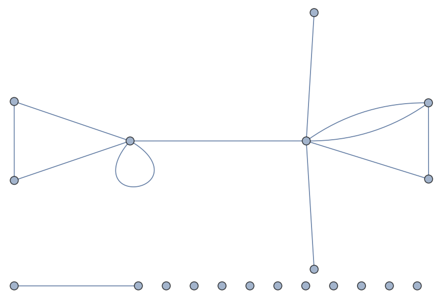
The available options are:

- `DirectedEdges → True` creates a directed graph.

In[206]:=

```
IGCallawayTraitsGame[20, 2, {1, 2, 3}, (1   0.5 0.3
                                        0.5 0.7 0.2
                                        0.3 0.2 0.1)]
```

Out[206]=



## IGEstablishmentGame

In[207]:=

```
? IGEstablishmentGame
```

IGEstablishmentGame[n, k, typeWeights, preferenceMatrix]

This function simulates a growing random graph according to the following algorithm:

At each time step, a new vertex is added. Its type is randomly selected according to the type weights. It attempts to connect to k distinct existing vertices. The probability of success for given types of vertices is given by the preference matrix.

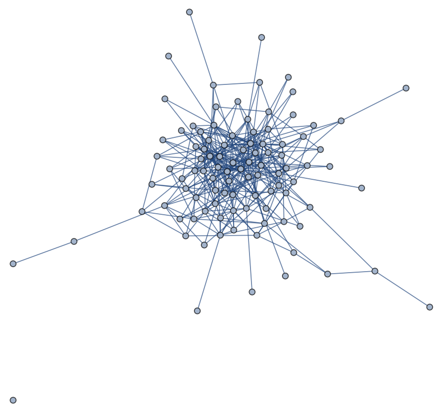
The available options are:

- `DirectedEdges → True` creates a directed graph.

In[208]:=

```
IGEstablishmentGame[100, 10, {2, 1, 2}, (1   0.5 0.3
                                         0.5 0.7 0.2
                                         0.3 0.2 0.1)]
```

Out[208]=



## IGGeometricGame

In[209]:=

```
? IGGeometricGame
```

IGGeometricGame[n, radius] generates an n–vertex
geometric random graph on the unit square by connecting points closer than radius.

Available options:

- "Periodic" → True assumes a toroidal topology

In[210]:=

```
IGGeometricGame[50, 0.2]
```

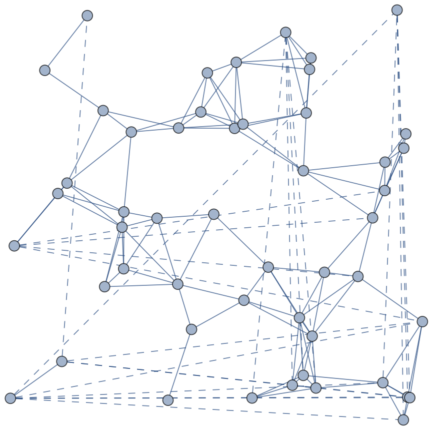
Out[210]=

Use a toroidal topology and draw “wraparound” edges with dashed lines.

In[211]:=

```
IGGeometricGame[50, 0.2, "Periodic" → True] //
 IGEdgeMap[
  If[EuclideanDistance @@ # > 0.2, Dashed, None] &, EdgeStyle → IGEdgeVertexProp[VertexCoordinates]
 ]
```

Out[211]=



# Graph modification

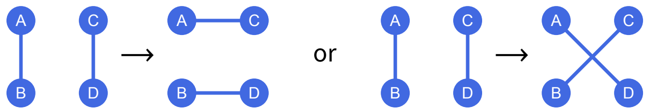
## IGRewire

In[212]:=

```
? IGRewire
```

IGRewire[graph, n] attempts to rewire the edges of graph n times
while preserving its degree sequence. Weights and other graph properties are discarded.

IGRewire will try to rewire the edges of the graph the given number of times by switching random pairs of edges as below, thus preserving the graph’s degree sequence.




The switches succeed only if they would not create multi-edges. The parameter *n* specifies the number of switch attempts, not the number of successful switches.

For directed graphs, the switches are such that they preserve both the in- and out-degree sequence.

The vertex ordering of the graph is retained.

**Warning:** Most graph properties, such as edge weights, will be lost.

The available options are:

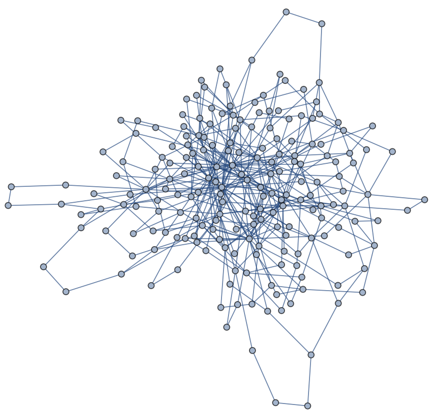
- `SelfLoops → True` allows the creation of self-loops.

Generate a random network with scale-free degree distribution:

In[213]:=

```
IGRewire[IGBarabasiAlbertGame[200, 2, DirectedEdges → False], 10 000]
```

Out[213]=

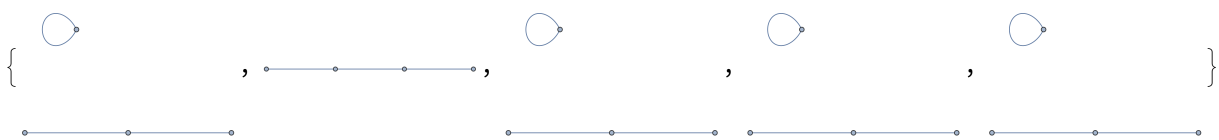


Use `SelfLoops → True` to allow creating loops.

In[214]:=

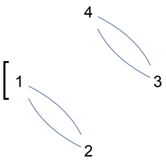
```
Table[IGRewire[PathGraph@Range[4], 100, SelfLoops → True], {5}]
```

Out[214]=



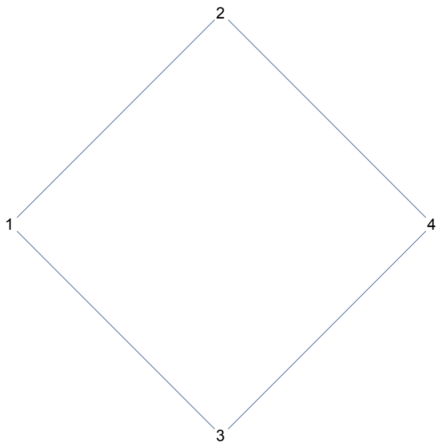
`IGRewrire` never creates any multi-edges. Multigraphs are allowed as input, but a warning is given.

In[215]:=

**IGRewire[**  **, 10, VertexShapeFunction → "Name"]**

IGRewire: The input is a multigraph. Multi-edges are never created during the rewiring process.

Out[215]=




Uniformly sample simple labelled graphs with a given degree sequence by first creating a single realization, then rewiring it a sufficient amount of times.

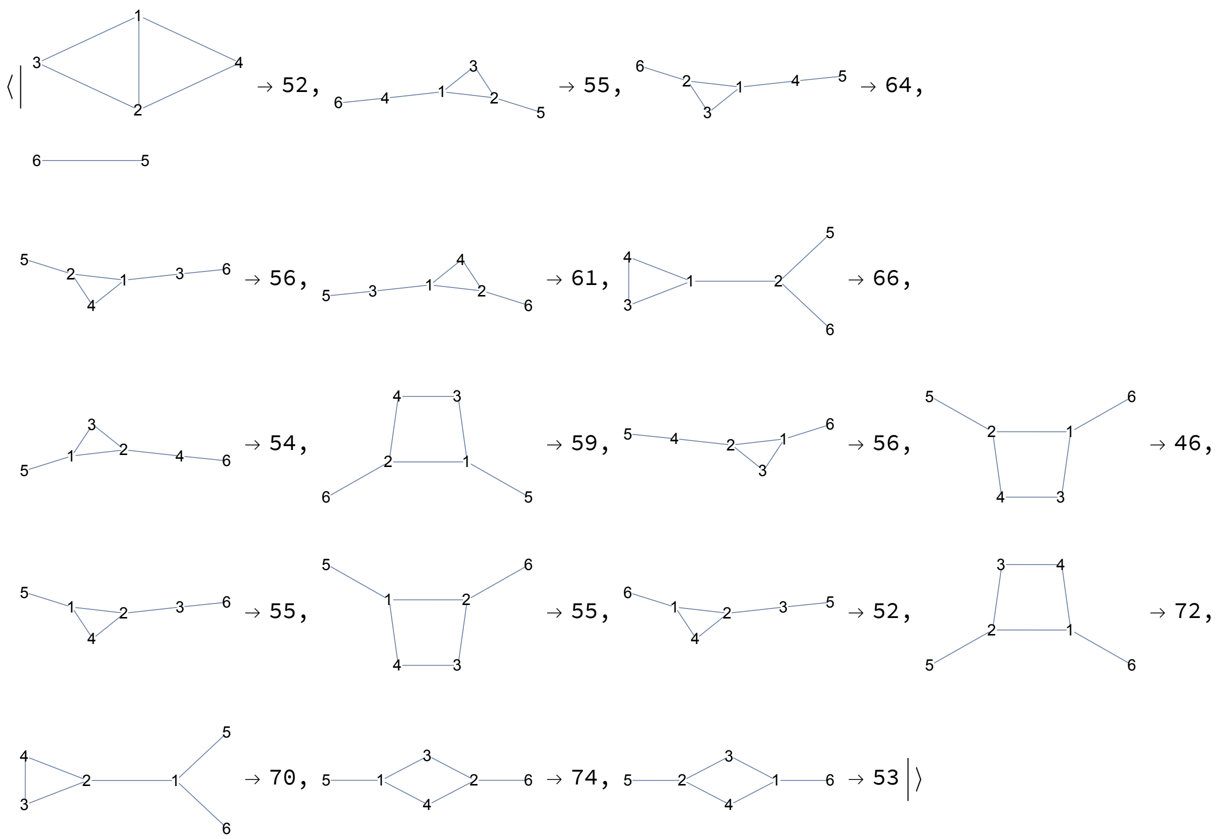
In[216]:=

```
degseq = {3, 3, 2, 2, 1, 1};
```

In[217]:=

```
Table[
    IGRewire[IGRealizeDegreeSequence[degseq], 100],
    {1000}
   ] // CountsBy[AdjacencyMatrix] // KeySort //
 KeyMap[AdjacencyGraph[#, VertexShapeFunction → "Name"] &]
```

Out[217]=




## IGRewireEdges

In[218]:=

```
? IGRewireEdges
```

IGRewireEdges[graph, p] rewires each edge of the graph with probability p. Weights and other graph properties are discarded.
IGRewireEdges[graph, p, "In"] rewires the starting point of each edge with probability p. The in–degree sequence is preserved.
IGRewireEdges[graph, p, "Out"] rewires the endpoint of each edge with probability p. The out–degree sequence is preserved.

`IGRewireEdges` randomly rewires each edge of the graph with the given probability. The vertex ordering is retained.

For directed graphs, it can optionally rewire only the starting point or endpoint of directed edges, thus preserving the out- or in-degree sequence. In this case, the `MultiEdges` option is ignored and multi-edges may be created.

**Warning:** Most graph properties, such as edge weights, will be lost.

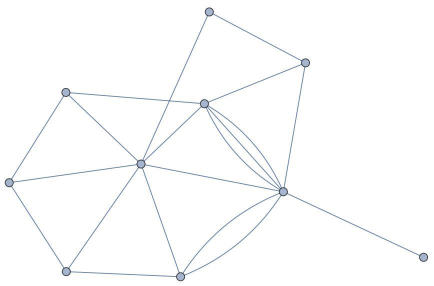
The available options are:

- `SelfLoops → True` allows the creation of self-loops.
- `MultiEdges → True` allows the creation of multi-edges.

Create a random graph with 10 vertices and 20 edges, while allowing for multi-edges:

In[219]:=
```
IGRewireEdges[RandomGraph[{10, 20}], 1, MultiEdges → True]
```

Out[219]=



In[220]:=
```
EdgeCount[%]
```

Out[220]=
```
20
```

Rewire the endpoint of each edge, preserving the out-degree sequence.

In[221]:=
```
g = RandomGraph[{10, 30}, DirectedEdges → True];
{VertexInDegree[g], VertexOutDegree[g]}
```

Out[222]=
```
{{5, 4, 1, 2, 2, 2, 5, 3, 3, 3}, {2, 6, 4, 2, 2, 4, 3, 2, 2, 3}}
```

In[223]:=
```
rg = IGRewireEdges[g, 1, "Out"];
{VertexInDegree[rg], VertexOutDegree[rg]}
```

Out[224]=
```
{{2, 0, 2, 7, 3, 3, 3, 3, 2, 5}, {2, 6, 4, 2, 2, 4, 3, 2, 2, 3}}
```

Note that multi-edges were created.

In[225]:=
```
MultigraphQ[rg]
```

Out[225]=
```
True
```

## IGVertexContract

In[226]:=
```
? IGVertexContract
```

IGVertexContract[g, {{v1, v2, …}, …}] returns a graph in which the specified vertex sets are contracted into single vertices.

`IGVertexContract[g, {set1, set2, …}]` will simultaneously contract multiple vertex sets into single vertices.

The name of a contracted vertex will be the same as the first element of the corresponding set. Vertex ordering is not retained. Edge ordering is retained only when using *both* `SelfLoops → True` and `MultiEdges → True`.

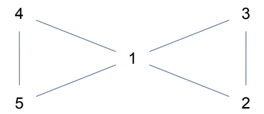
**Warning:** Most graph properties, such as edge weights, will be lost.

The available options are:

- `SelfLoops → True` keeps any self-loops created during contraction.
- `MultiEdges → True` keeps any parallel edges created during contraction.

In[227]:=

```
g = 
```




```
;
```

In[228]:=

```
IGVertexContract[g, {{1, 2, 3}, {4, 5}}, VertexLabels → "Name"]
```

Out[228]=


In[229]:=

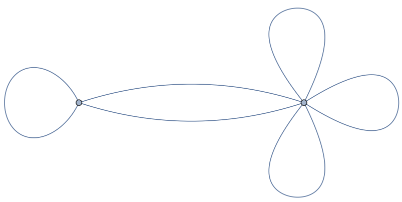
```
IGVertexContract[g, {{1, 2, 3}, {4, 5}}, SelfLoops → True]
```

Out[229]=

In[230]:=

```
IGVertexContract[g, {{1, 2, 3}, {4, 5}}, SelfLoops → True, MultiEdges → True]
```

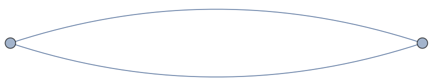
Out[230]=



In[231]:=

```
IGVertexContract[g, {{1, 2, 3}, {4, 5}}, MultiEdges → True]
```

Out[231]=

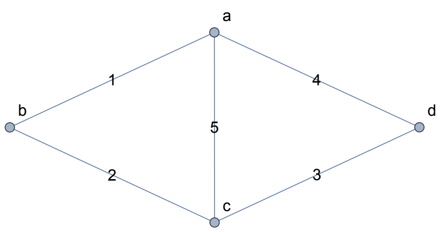


When using both `SelfLoops → True` and `MultiEdges → True`, the edge ordering is maintained relative to the input graph. This allows easily transferring edge weights, and combining them if necessary.

In[232]:=

```
g = IGShorthand["a-b-c-d-a,a-c",
  EdgeWeight → {1, 2, 3, 4, 5}, EdgeLabels → "EdgeWeight"]
```

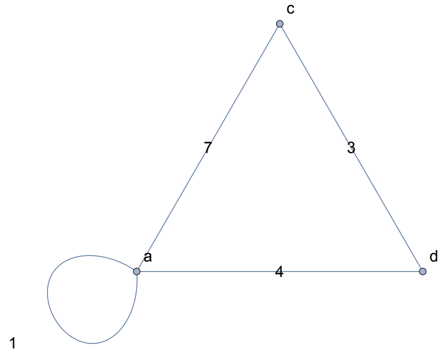
Out[232]=



In[233]:=

```
IGWeightedSimpleGraph[
 IGVertexContract[g, {{"a", "b"}},
  SelfLoops → True, MultiEdges → True,
  EdgeWeight → IGEdgeProp[EdgeWeight][g]
 ],
 EdgeLabels → "EdgeWeight", VertexLabels → "Name"
]
```

Out[233]=




## IGConnectNeighborhood

In[234]:=

```
? IGConnectNeighborhood
```

IGConnectNeighborhood[graph] connects each vertex in graph to its 2nd order neighbourhood.
IGConnectNeighborhood[graph, k] connects each vertex in graph to its order k neighbourhood. Weights and other graph properties are discarded.

`IGConnectNeighborhood[g, k]` connects each vertex in g to its order k neighbourhood. This operation is also known as the $k^{\text{th}}$ power of the graph.

`IGConnectNeighborhood` differs from the built-in `GraphPower` in that it preserves parallel edges and self-loops.

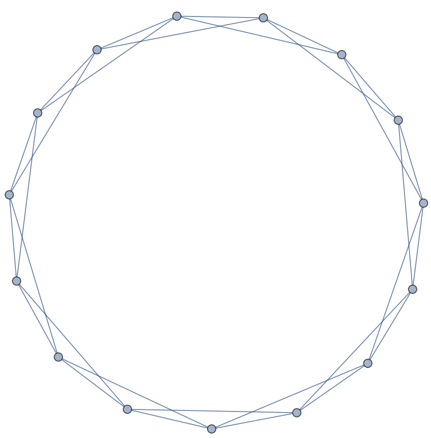
**Warning:** Most graph properties, such as edge weights, will be lost.

Connect each vertex to its second order neighbourhood:

In[235]:=

```
IGConnectNeighborhood[CycleGraph[15]]
```

Out[235]=

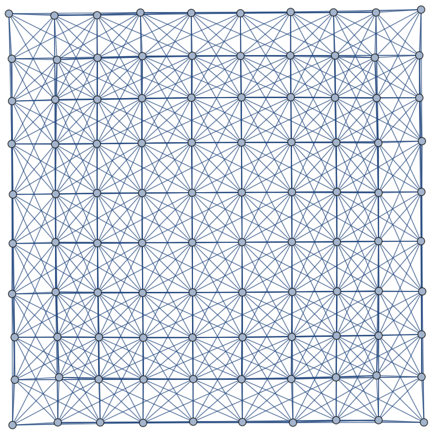


Connect each vertex to its third order neighbourhood:

In[236]:=

```
IGConnectNeighborhood[GridGraph[{10, 10}], 3]
```

Out[236]=



## IGMycielskian

In[237]:=

```
? IGMycielskian
```

IGMycielskian[graph] returns the Mycielskian of graph.

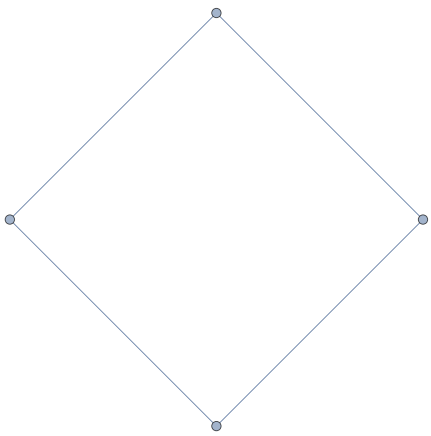
`IGMycielskian` applies the Mycielski construction to an undirected graph on $n \geq 2$ vertices to obtain a larger graph (the *Mycielskian*) on $2n + 1$ vertices. If the graph has less than 2 vertices, then instead of applying the standard Mycielski construction, `IGMycielskian` simply adds one vertex and one edge.

If the original graph has chromatic number $k$, its Mycielskian has chromatic number $k + 1$. The Mycielski construction preserves the triangle-free property of the graph.

In[238]:=

```
g = CycleGraph[4]
```

Out[238]=



In[239]:=

```
{IGChromaticNumber[g], IGTriangleFreeQ[g]}
```

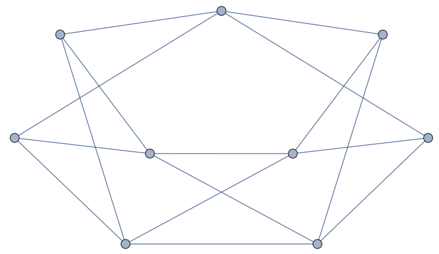
Out[239]=

```
{2, True}
```

In[240]:=

```
mg = IGMycielskian[g]
```

Out[240]=



In[241]:=

```
{IGChromaticNumber[mg], IGTriangleFreeQ[mg]}
```

Out[241]=

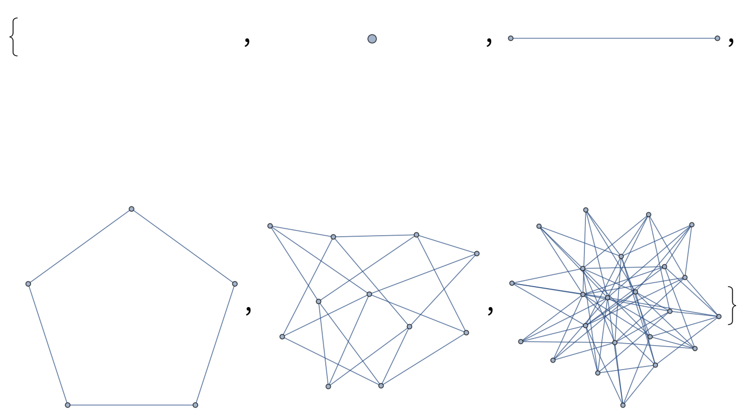
```
{3, True}
```

Construct triangle-free graphs with successively larger chromatic numbers.

In[242]:=

```
NestList[IGMycielskian, IGEmptyGraph[], 5]
```

Out[242]=



In[243]:=

```
IGChromaticNumber /@ %
```

Out[243]=

```
{0, 1, 2, 3, 4, 5}
```

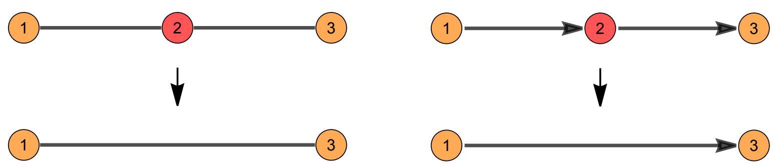
## IGSmoothen

In[244]:=

```
? IGSmoothen
```

IGSmoothen[graph] suppresses degree−2 vertices, thus obtaining the smallest topologically equivalent graph. The weights of merged edges are added up.

`IGSmoothen` suppresses all degree-2 vertices, thus obtaining the smallest topologically equivalent (i.e. homeomorphic) graph. See also `IGHomeomorphicQ`.



The vertex names are preserved, and the weights of merged edges are summed up. All other graph properties are discarded. In directed graphs, only those vertices are smoothened which have one incoming and one outgoing edge.

Available options:

- `DirectedEdges → False` ignores edge directions in the input graph.

The smallest topological equivalent of a path graph consists of two connected vertices.

In[245]:=

`IGSmoothen[`   `, VertexLabels → Automatic]`

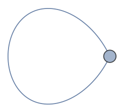
Out[245]=


The result may contain self-loops. The smallest topological equivalent of a cycle graph is a single vertex with a self-loop.

In[246]:=

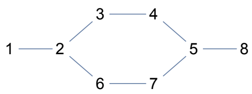
```
IGSmoothen[CycleGraph[10]]
```

Out[246]=



The result may also contain multi-edges.

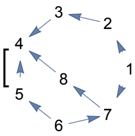
In[247]:=

`IGSmoothen[`   `, VertexLabels → Automatic]`

Out[247]=

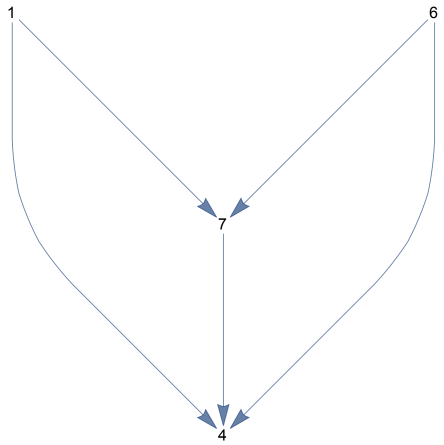
If the input is directed, only those vertices are smoothed which have one incoming and one outgoing edge.

In[248]:=

**IGSmoothen[**

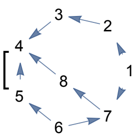
**, VertexShapeFunction → "Name"]**

Out[248]=



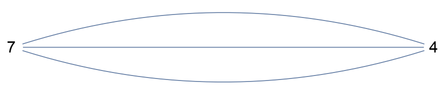

Use `DirectedEdges → False` to treat the input graph as undirected.

In[249]:=

**IGSmoothen[**

**, DirectedEdges → False, VertexShapeFunction → "Name"]**

Out[249]=

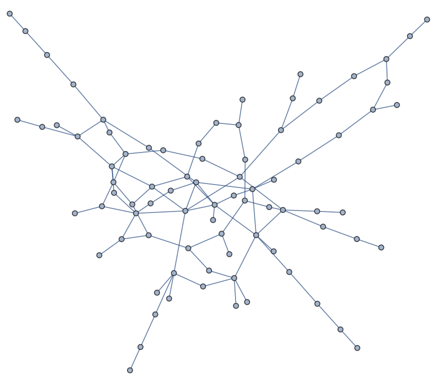



The result is always a weighted graph. When contracting edges, their weights are added up. If the input graph was not weighted, all of its edge weights are considered to be 1. Thus, the graph distance of any two vertices in the result is always the same as it was in the input graph.

In[250]:=

```
g = IGGiantComponent@RandomGraph[{100, 100}]
```

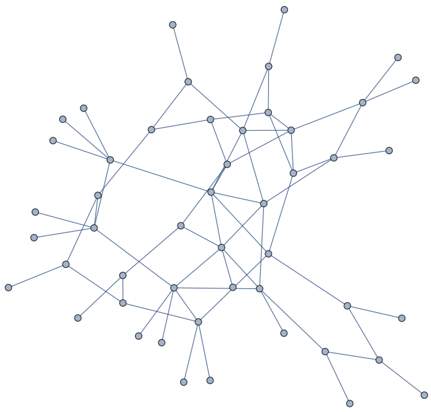
Out[250]=



In[251]:=

```
tm = IGSmoothen[g]
```

Out[251]=



In[252]:=

```
IGEdgeWeightedQ[tm]
```

Out[252]=

```
True
```

In[253]:=

```
IGDistanceMatrix[g, VertexList[tm], VertexList[tm]] == IGDistanceMatrix[tm]
```

Out[253]=

```
True
```

The result does not contain any degree-2 vertices, except possibly isolated vertices with self-loops.

In[254]:=

```
Union@VertexDegree[tm]
```

Out[254]=

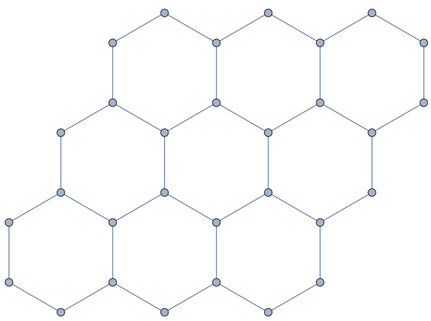
```
{1, 3, 4, 5, 6}
```

The vertex coordinates, as well as any other graph properties are discarded.

In[255]:=

```
g = IGMeshGraph@IGLatticeMesh["Hexagonal", {3, 3}]
```

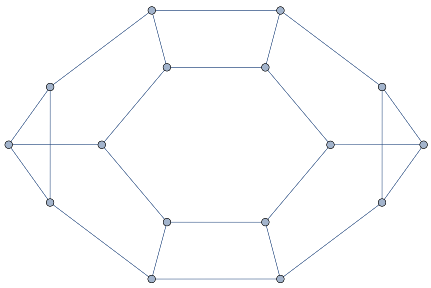
Out[255]=



In[256]:=

```
IGSmoothen[g]
```

Out[256]=

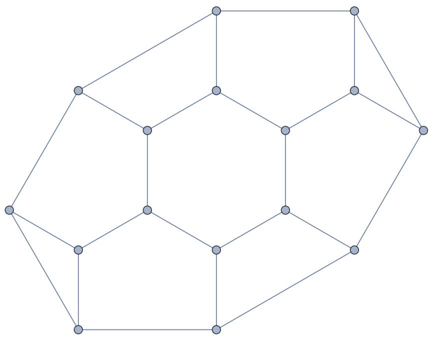


Vertex coordinates can be transferred to the new graph as follows:

In[257]:=

```
IGSmoothen[g,
 VertexCoordinates → {v_ :→ PropertyValue[{g, v}, VertexCoordinates]}]
```

Out[257]=

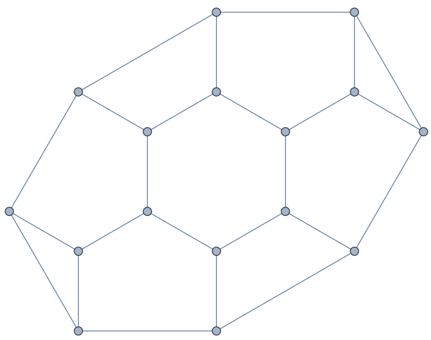


An alternative and faster method uses `IGVertexMap` and `IGVertexAssociate`:

In[258]:=

```
IGSmoothen[g] // IGVertexMap[IGVertexAssociate[GraphEmbedding][g], VertexCoordinates → VertexList]
```

Out[258]=

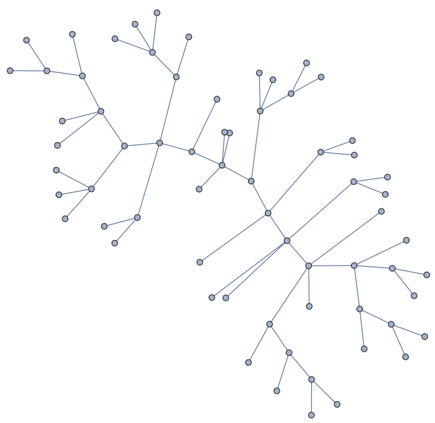


Create a tree in which every non-leaf node has a degree of at least 3.

In[259]:=

```
IGSmoothen[IGTreeGame[100], GraphLayout → "RadialEmbedding"]
```

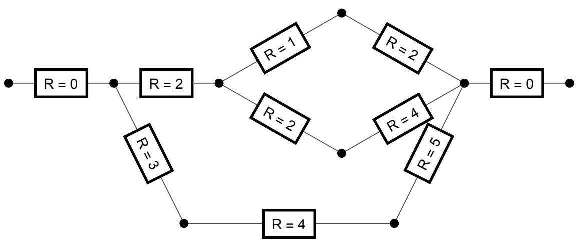
Out[259]=



Let us compute the effective resistance of a resistor network by repeated smoothing (merger of resistors in series) and simplification (merger of resistors in parallel). Resistances are stored as edge weights. A zero-resistance input and output terminal is added to prevent the premature smoothing of these points.

In[260]:=

```
resistorGrid =
```



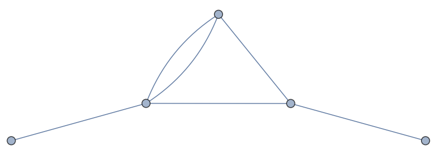

```
;
```

Merge resistors in series ...

In[261]:=

```
reducedGrid = IGSmoothen[resistorGrid]
```

Out[261]=

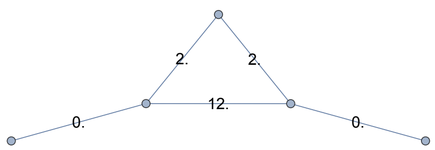


... then merge resistors in parallel and check the resulting edge weights.

In[262]:=

```
reducedGrid = IGWeightedSimpleGraph[reducedGrid, 1 / Total[1 / {##}] &, EdgeLabels → "EdgeWeight"]
```

Out[262]=



Repeat until a single resistor remains.

In[263]:=

```
reducedGrid = IGSmoothen[reducedGrid]
```

Out[263]=

In[264]:=

```
reducedGrid = IGWeightedSimpleGraph[reducedGrid, 1 / Total[1 / {##}] &, EdgeLabels → "EdgeWeight"]
```

Out[264]=




In[265]:=

```
reducedGrid = IGSmoothen[reducedGrid, EdgeLabels → "EdgeWeight"]
```

Out[265]=




In[266]:=

```
IGEdgeProp[EdgeWeight][reducedGrid]
```

Out[266]=

```
{3.}
```

# Structural properties

## Centrality measures

Centralities are various measures that quantify the “importance” of vertices or edges in graphs.

### Betweenness

In[267]:=

```
? IGBetweenness
```

IGBetweenness[graph] gives a list of betweenness centralities for the vertices of graph.
IGBetweenness[graph, {vertex1, vertex2, …}] gives a list of betweenness centralities for the specified vertices.

In[268]:=

```
? IGBetweennessCutoff
```

IGBetweennessCutoff[graph, cutoff] gives the range−limited betweenness centralities by considering only paths of at most length cutoff.
IGBetweennessCutoff[graph, cutoff, {vertex1, vertex2, …}] gives the range−limited betweenness centralities for the specified vertices.

In[269]:=

```
? IGEdgeBetweenness
```

IGEdgeBetweenness[graph] gives a list of betweenness centralities for the edges of graph.

In[270]:=

```
? IGEdgeBetweennessCutoff
```

IGEdgeBetweennessCutoff[graph, cutoff] gives the range−limited edge betweenness centralities by considering only paths of at most length cutoff.

The betweenness of a vertex or edge is, roughly speaking, the number of shortest paths passing through it. More formally, the betweenness of vertex $i$ is $b_i = \sum_{i \neq s \neq t} \frac{g_{st}^{(i)}}{g_{st}}$, where $g_{st}$ is the total number of shortest paths (geodesics) between vertices $s$ and $t$, and $g_{st}^{(i)}$ is the number of shortest paths between vertices $s$ and $t$ that pass through $i$.

Weighted graphs and multigraphs are supported by all betweenness functions in IGraph/M.

Note that as of *Mathematica* 13.0, the built-in `BetweennessCentrality` function ignores edge weights and multi-edges, which causes it to yield different results from `IGBetweenness`.

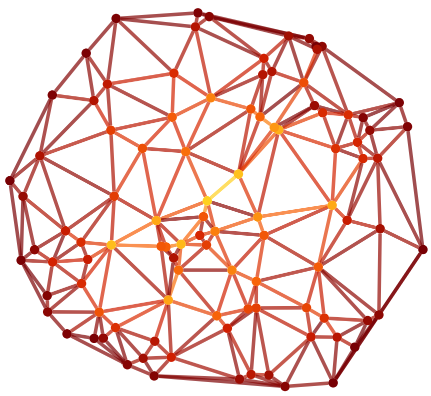
Available options:

- `Normalized → True` will compute the normalized betweenness by dividing the result by the number of (ordered or unordered) vertex pairs used in the shortest path calculation. Thus the normalization factor is $(V-1)(V-2)$ for directed graphs and $\frac{1}{2}(V-1)(V-2)$ for undirected graphs. The normalized value lies between 0 and 1.

Visualize the vertex and edge betweenness of a weighted geometrical graph, where weights represent Euclidean distances.

In[271]:=
```
pts = RandomPoint[Disk[], 100];
IGMeshGraph[
  DelaunayMesh[pts],
  EdgeStyle → Thick, VertexStyle → EdgeForm[None]
 ] //
 IGVertexMap[
   ColorData["SolarColors"],
   VertexStyle → Rescale@*IGBetweenness
  ]/*
  IGEdgeMap[
   ColorData["SolarColors"],
   EdgeStyle → Rescale@*IGEdgeBetweenness
  ]
```

Out[272]=



Compute the betweenness of a subset of vertices.

In[273]:=
```
g = ExampleData[{"NetworkGraph", "DolphinSocialNetwork"}];
```

In[274]:=
```
Take[VertexList[g], 5]
```

Out[274]=
```
{Beak, Beescratch, Bumper, CCL, Cross}
```

In[275]:=

```
IGBetweenness[g, %]
```

Out[275]=

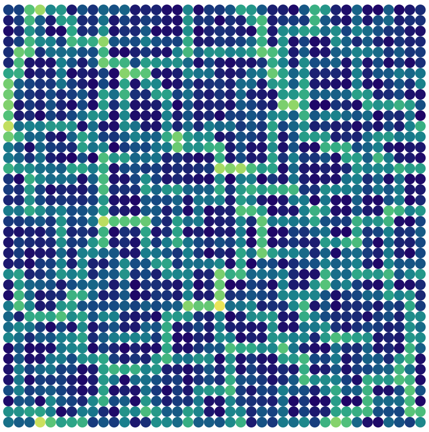
```
{34.9212, 390.384, 16.6032, 4.34405, 0.}
```

Visualize the betweenness of a periodic grid with slightly randomized edge weights.

In[276]:=

```
n = 40;
IGSquareLattice[{n, n},
  "Periodic" → True,
  VertexCoordinates → Tuples[Range[n], {2}],
  EdgeWeight → {_ :→ RandomReal[{.99, 1.01}]},
  GraphStyle → "BasicBlack",
  EdgeShapeFunction → None,
  VertexSize → 1
 ] // IGVertexMap[
  ColorData["BlueGreenYellow"],
  VertexStyle → Rescale@*IGBetweenness
 ]
```

Out[277]=



### Possible issues

Betweenness computation involves comparing the lengths of paths, and deciding which specific path is the shortest, and which paths have equal lengths. When non-integer edge weights are used, the path length computation is subject to roundoff errors, which may cause the path length comparison to fail. igraph mitigates this by comparing the lengths using tolerances, however, there is still a small risk that roundoff errors may affect the result. To avoid this potential problem entirely, use integer weights. For example, if the weights are rational, multiply them by the least common multiple of their denominators.

## Closeness

In[278]:=

```
? IGCloseness
```

IGCloseness[graph] gives a list of closeness centralities for the vertices of graph.
IGCloseness[graph, {vertex1, vertex2, …}] gives a list of closeness centralities for the specified vertices.

In[279]:=

```
? IGClosenessCutoff
```

IGClosenessCutoff[graph, cutoff] gives the range−limited closeness centralities by considering only paths of at most length cutoff.
IGClosenessCutoff[graph, cutoff, {vertex1, vertex2, …}] gives the range−limited closeness centralities for the specified vertices.

In[280]:=

```
? IGNeighborhoodCloseness
```

IGNeighborhoodCloseness[graph, cutoff] gives the range−limited closeness centralities along with the number of vertices reachable within the cutoff distance.
IGNeighborhoodCloseness[graph, cutoff, {vertex1, vertex2, …}] gives the range−limited closeness centralities and number of reachable vertices for the specified vertices.

The normalized closeness centrality of a vertex is the inverse average shortest path length to other vertices.

Weighted graphs are supported.

Available options:

- Normalized → False will compute the non-normalized closeness, i.e. the inverse of the sum of shortest path lengths to all other vertices.

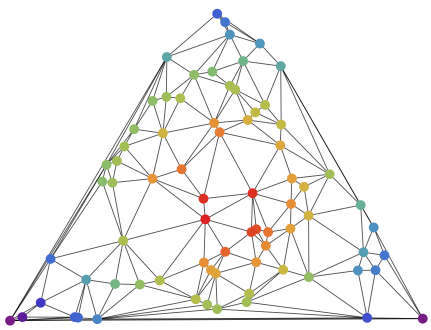
There is no standard definition of closeness centrality for disconnected graphs. When the graph is disconnected, IGraph/M will only consider the distances to reachable vertices. In the undirected case, this effectively computes the closeness separately for each connected component. Use IGNeighborhoodCloseness to obtain both the closeness values, as well as how many vertices were reachable from each vertex. This information allows for computing various generalizations of closeness centrality for disconnected graphs.

Visualize the closeness of nodes in a weighted geometrical graph where weights correspond to Euclidean distances.

In[281]:=

```
pts = RandomPoint[Polygon@CirclePoints[3], 75];
IGVertexMap[
 ColorData["Rainbow"],
 VertexStyle → Rescale@*IGCloseness,
 IGMeshGraph[DelaunayMesh[pts], GraphStyle → "BasicBlack"]
]
```

Out[282]=



For isolated vertices, Indeterminate is returned.

In[283]:=

```
IGCloseness@IGShorthand["1,2-3"]
```

Out[283]=

```
{Indeterminate, 1., 1.}
```

## Harmonic centrality

In[284]:=

```
? IGHarmonicCentrality
```

IGHarmonicCentrality[graph] gives the harmonic centralities for the vertices of graph.
IGHarmonicCentrality[graph, {vertex1, vertex2, …}] gives the harmonic centralities for the specified vertices.

In[285]:=

```
? IGHarmonicCentralityCutoff
```

IGHarmonicCentralityCutoff[graph, cutoff] gives the range−limited harmonic centralities by considering only paths of at most length cutoff.
IGHarmonicCentralityCutoff[graph, cutoff, {vertex1, vertex2, …}] gives the range−limited harmonic centralities of the specified vertices.

The harmonic centrality of a vertex is the average inverse shortest path length to all other vertices. The inverse shortest path length to unreachable vertices is considered to be zero.

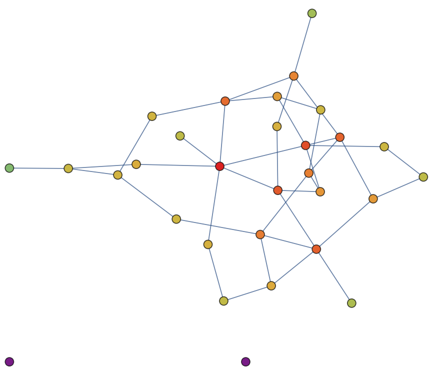
Available options:

- `Normalized → False` computes the non-normalized harmonic centrality, i.e. the sum of inverse shortest path length to all other vertices.

In[286]:=

```
RandomGraph[{30, 40}, VertexSize → Large] //
 IGVertexMap[ColorData["Rainbow"], VertexStyle → IGHarmonicCentrality /* Rescale]
```

Out[286]=



## PageRank

In[287]:=

```
? IGPageRank
```

IGPageRank[graph] gives a list of PageRank centralities for the vertices of the graph using damping factor 0.85. Available Method options: {"Arnoldi", "PRPACK"}.
IGPageRank[graph, damping] gives a list of PageRank centralities for the vertices of the graph using the given damping factor.

In[288]:=

```
? IGPersonalizedPageRank
```

IGPersonalizedPageRank[graph, reset] gives a list of personalized PageRank centralities for the vertices of the graph with personalization vector reset.
IGPersonalizedPageRank[graph, reset, damping] uses the given damping factor.
IGPersonalizedPageRank[graph, <| vertex1 -> weight1, vertex2 -> weight2, … |>, damping] uses non–zero personalization weights only for the specified vertices.

The PageRank centrality of a vertex is the fraction of time a random walker would spend on that vertex. The walker jumps from vertex to vertex randomly, following outward edges with probabilities proportional to their weights. Additionally, after each step, with a probability $1 - d$ the walk is restarted from a random vertex. $d$ is called the damping factor. If the walker is stuck in a sink vertex (i.e. a vertex with no outgoing edges), the walk is also restarted.

In the standard version of PageRank, when the walk is restarted, the starting vertex is chosen uniformly. In the personalized version, it is chosen with probabilities proportional to the values in the `reset` parameter.

Weighted graphs and multigraphs are supported, and self-loops are taken into consideration.

Note that as of Mathematica 13.0, the built-in `PageRankCentrality` function ignores self-loops.

The default damping factor is 0.85.

The following `Method` options are available:

- `"Arnoldi"` uses ARPACK, and solves PageRank as an eigenvalue problem.
- `"PRPACK"` uses PRPACK and uses the algebraic method. It is the default method, and usually much faster than `"Arnoldi"`.

Plot the logarithmic histogram of PageRank scores of the network of webpage in the nd.edu domain.

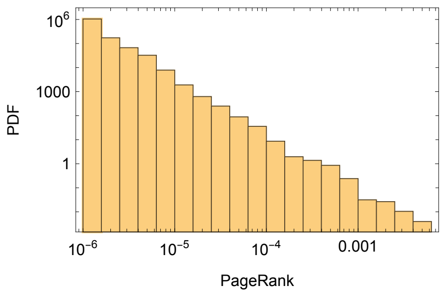
In[289]:=

```
ndWeb = ExampleData[{"NetworkGraph", "WorldWideWeb"}];
```

In[290]:=

```
Histogram[IGPageRank[ndWeb], "Log", {"Log", "PDF"},
 Frame → True, FrameLabel → {"PageRank", "PDF"}]
```

Out[290]=




The personalization weights may be given as a vector of the same length as the vertex list …

In[291]:=

```
g = RandomGraph[{10, 20}, DirectedEdges → True];
IGPersonalizedPageRank[g, RandomReal[1, VertexCount[g]]]
```

Out[292]=

```
{0.0433579, 0.129471, 0.23356, 0.113496,
 0.000101892, 0.12872, 0.260584, 0.024101, 0.0324582, 0.0341504}
```

… or as an association from vertex names to weights, in which case the weight of non-included vertices is taken to be zero.

In[293]:=

```
IGPersonalizedPageRank[g, <|1 → 1.5, 3 → 0.5|>]
```

Out[293]=

```
{0.297935, 0.0717967, 0.168933, 0.290878, 0., 0.0376334, 0.132824, 0., 0., 0.}
```

Personalize PageRank by always restarting the walk from one of two vertices (29 or 74) on a grid graph:

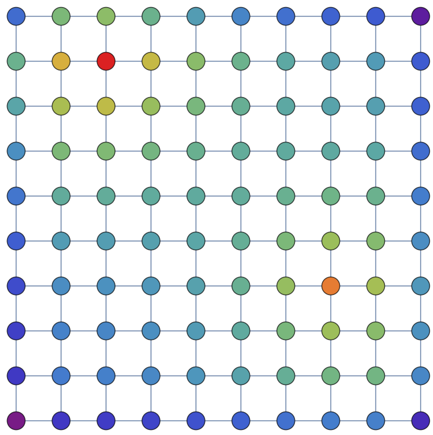
In[294]:=

```
g = IGSquareLattice[{10, 10}, VertexSize → Large];
```

In[295]:=

```
g // IGVertexMap[ColorData["Rainbow"],
  VertexStyle → (IGPersonalizedPageRank[#, <|29 → 1, 74 → 1|>, 0.99] &/*Rescale)]
```

Out[295]=



## LinkRank

In[296]:=

```
? IGLinkRank
```

IGLinkRank[graph] gives a list of LinkRank centralities for the edges of
 the graph using damping factor 0.85. Available Method options: {"Arnoldi", "PRPACK"}.
IGLinkRank[graph, damping] gives a list of LinkRank centralities for the edges of the graph using the given damping factor.

In[297]:=

```
? IGPersonalizedLinkRank
```

IGPersonalizedLinkRank[graph, reset] gives a list of personalized
 LinkRank centralities for the edges of the graph with personalization vector reset.
IGPersonalizedLinkRank[graph, reset, damping] uses the given damping factor.
IGPersonalizedLinkRank[graph, <| vertex1 -> weight1, vertex2 ->
 weight2, … |>, damping] uses non-zero personalization weights only for the specified vertices.

LinkRank is the equivalent of PageRank for edges. The LinkRank of an edge is the relative frequency of traversing that edge by a random walker. For a detailed description of the random walk process, see the PageRank section.

The LinkRank of edges can be computed from the PageRank by simply dividing the PageRank of each vertex between its outgoing edges, proportionally with their edge weights. The LinkRank scores of the out-edges of a vertex add up to the PageRank of that vertex. The LinkRank scores of all edges in the graph add up to 1.

Weighted graphs and multigraphs are supported, and self-loops are taken into consideration.

The available `Method` options are the same as for `IGPageRank`.

Visualize both the LinkRank and PageRank of a random directed graph.

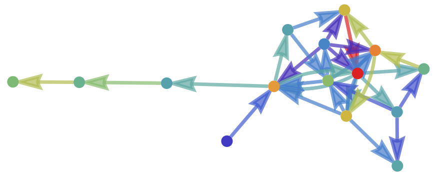
In[298]:=

```
maxNorm = # / Max[#] &;

g = RandomGraph[{15, 30}, DirectedEdges → True,
   EdgeStyle → Thick, VertexSize → Large, GraphStyle → "BasicBlack"];

g // IGEdgeMap[ColorData["Rainbow"], EdgeStyle → IGLinkRank /* maxNorm] //
 IGVertexMap[ColorData["Rainbow"], VertexStyle → IGPageRank /* maxNorm]
```

Out[300]=

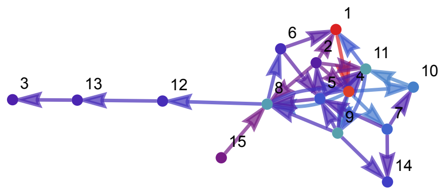


Visualize the personalized version of LinkRank and PageRank, always starting the random walk from vertex 1.

In[301]:=

```
pers = <|1 → 1|>;
Graph[g, VertexLabels → "Name"] //
  IGEdgeMap[ColorData["Rainbow"], EdgeStyle → (IGPersonalizedLinkRank[#, pers] &) /* maxNorm] //
 IGVertexMap[ColorData["Rainbow"], VertexStyle → (IGPersonalizedPageRank[#, pers] &) /* maxNorm]
```

Out[302]=




## Eigenvector centrality

In[303]:=

```
? IGEigenvectorCentrality
```

IGEigenvectorCentrality[graph] gives the eigenvector centrality of each vertex.

Eigenvector centrality is based on the idea that the importance (centrality) of a vertex should be affected not only by how many other vertices point to it, but also by the importance of its neighbours. The eigenvector centrality of a vertex is proportional to the sum of centralities of its neighbours. Mathematically, the eigenvector centrality is the leading eigenvector of the adjacency matrix.

Eigenvector centrality is meaningful for connected graphs only. Disconnected graphs should be decomposed into their components, and the eigenvector centrality computed separately for each. The vertex centrality scores will be comparable only within components, not between separate components.

In undirected graphs, the diagonal of the adjacency matrix is assumed to contain *twice* the number of self-loops on each vertex. This makes the undirected result consistent with the directed one when each undirected edge is replaced by reciprocal directed ones.

For directed graphs, the left eigenvector of the adjacency matrix is calculated. In other words, the centrality of a vertex is proportional to the sum of centralities of vertices pointing *to* it.

Weighted and directed graphs are supported.

The available options are:

- `Normalized → True` will scale the result so that the maximum centrality is 1. The default is `True`.
- `DirectedEdges → False` ignores edge directions.

### Kleinberg’s hub and authority scores

In[304]:=

```
? IGHubScore
```

IGHubScore[graph] gives Kleinberg's hub score for each vertex.

In[305]:=

```
? IGAuthorityScore
```

IGAuthorityScore[graph] gives Kleinberg's authority score for each vertex.

Weighted graphs are supported.

The available options are:

- `Normalized → True` scales the result so that the maximum centrality is 1. The default is `True`.

### Burt’s constraint score

In[306]:=

```
? IGConstraintScore
```

IGConstraintScore[graph] returns Burt's constraint score for each vertex.

Weighted graphs are supported.

## Centralization

In[307]:=

```
? IG*Centralization
```

▼ IGraphM`

| | |
|---|---|
| IGBetweennessCentralization | IGDegreeCentralization |
| IGClosenessCentralization | IGEigenvectorCentralization |

*Centralization* is computed from centrality values in a way equivalent to `Total[Max[centralities] - centralities]`. With the (default) option `Normalized → True`, the result is normalized by dividing by the highest possible centralization score of any graph of the same directedness on the same number of vertices.

In[308]:=

```
g = IGBarabasiAlbertGame[100, 2, DirectedEdges → False];
```

In[309]:=

```
IGBetweennessCentralization[g]
```

Out[309]=

```
0.194343
```

In[310]:=

```
IGClosenessCentralization[g]
```

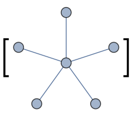
Out[310]=

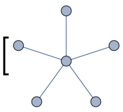
```
0.275726
```

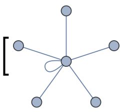
In[311]:=
```
IGDegreeCentralization[g, SelfLoops → False]
```
Out[311]=
```
0.144919
```
In[312]:=
```
IGEigenvectorCentralization[g]
```
Out[312]=
```
0.820631
```
For most centrality types, the highest centralization is achieved by the `StarGraph`.

In[313]:=
```
IGBetweennessCentralization@StarGraph[5]
```
Out[313]=
```
1.
```
In the case of the degree centralization, the highest possible centralization score depends on whether self-loops are allowed. This is controlled by the `SelfLoops` option of `IGDegreeCentralization`. The default is `SelfLoops → True`.

In[314]:=

{IGDegreeCentralization[  ],

IGDegreeCentralization[  , SelfLoops → False],

IGDegreeCentralization[  ]}

Out[314]=
```
{0.666667, 1., 1.}
```
## Topological sorting and acyclic graphs

### IGDirectedAcyclicGraphQ

In[315]:=
```
? IGDirectedAcyclicGraphQ
```
IGDirectedAcyclicGraphQ[graph] tests if graph is directed and acyclic.

`IGDirectedAcyclicGraphQ` tests if a graph is directed and has no directed cycles.

In[316]:=
```
IGDirectedAcyclicGraphQ /@ {IGShorthand["1->2->3->1"], IGShorthand["1->2->3<-1"]}
```
Out[316]=
```
{False, True}
```
`IGDirectedAcyclicGraphQ` returns `True` for graphs with no edges.

In[317]:=
```
IGDirectedAcyclicGraphQ[IGEmptyGraph[3]]
```
Out[317]=
```
True
```

## IGTopologicalOrdering

In[318]:=

```
? IGTopologicalOrdering
```

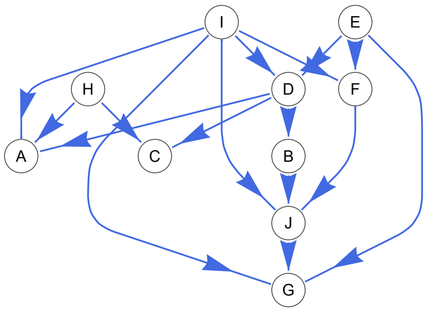
IGTopologicalOrdering[graph] returns a permutation that sorts the vertices in topological order. Note that the values returned are vertex indices, not vertex names.

`IGTopologicalOrdering` is to the built-in `TopologicalSort` as `Ordering` is to `Sort`: it returns the permutation which sorts vertices in topological order. When vertices are ordered topologically, all directed edges point from earlier vertices to later ones.

Graphs must be acyclic for topological sorting to be possible.

In[319]:=

```
g =
```




```
;
```

In[320]:=

```
IGDirectedAcyclicGraphQ[g]
```

Out[320]=

```
True
```

In[321]:=

```
p = IGTopologicalOrdering[g]
```

Out[321]=

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

In[322]:=

```
VertexList[g][[p]]
```

Out[322]=

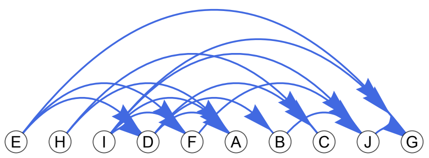
```
{E, H, I, D, F, A, B, C, J, G}
```

If the vertices are laid out from left to right in topological order, all edges will point from left to right.

In[323]:=

```
curvedEdge[offset_][{a_, ___, b_}, ___] :=
 Arrow@BezierCurve[{a, (a + b) / 2 + offset Reverse[b - a], b}]
Graph[g,
  EdgeShapeFunction → curvedEdge[2 / 3]
  (* in M12.0 or later simply use {{"CurvedEdge","Curvature"→1.5}} *)
 ] // IGVertexMap[{#, 0} &, VertexCoordinates → IGTopologicalOrdering /* Ordering]
```

Out[324]=




When the graph contains cycles, `$Failed` is returned.

In[325]:=

```
IGTopologicalOrdering[IGShorthand["1->2->3->4->5, 5->3, 5->6"]]
```

IGraphM: src/properties/dag.c:114 – The graph has cycles; topological sorting is only possible in acyclic graphs.

IGraphM: igraph returned with error: Invalid value.

Out[325]=

```
$Failed
```

## IGFeedbackArcSet

In[326]:=

```
? IGFeedbackArcSet
```

IGFeedbackArcSet[graph] computes a feedback edge set of graph. Removing these edges makes the graph acyclic. Available Method options: {"IntegerProgramming", "EadesLinSmyth"}. "IntegerProgramming" is guaranteed to find a minimum feedback arc set.

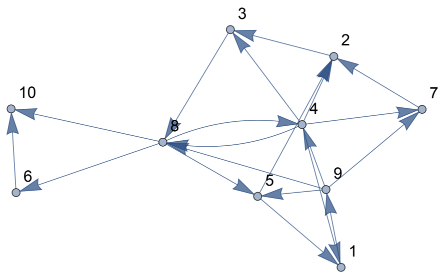
`IGFeedbackArcSet[]` returns a set of directed edges (also called *arcs*) the removal of which makes the graph acyclic. With `Method → "IntegerProgramming"`, it finds an exact minimal feedback arc set through integer programming using the triangle inequality formulation. With `Method → "EadesLinSmyth"`, it finds a feedback arc set (not necessarily minimal) using the fast “GR” heuristic of Eades, Lin and Smyth (1993).

The following directed graph is not acyclic.

In[327]:=

```
g = RandomGraph[{10, 20}, DirectedEdges → True, VertexLabels → "Name"]
```

Out[327]=




In[328]:=

```
{AcyclicGraphQ[%], IGDirectedAcyclicGraphQ[%]}
```

Out[328]=

```
{False, False}
```

Find a set of edges whose removal breaks all cycles.

In[329]:=

```
IGFeedbackArcSet[g]
```

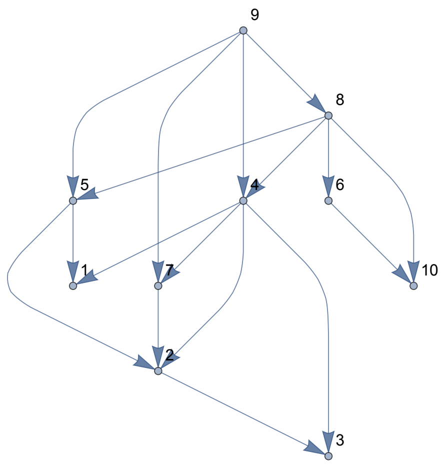
Out[329]=

```
{1 ↔ 9, 3 ↔ 8, 4 ↔ 8}
```

In[330]:=

```
ag = EdgeDelete[g, %]
```

Out[330]=




In[331]:=

```
IGDirectedAcyclicGraphQ[ag]
```

Out[331]=

```
True
```

Vertices of a directed acyclic graph can be sorted topologically. `IGTopologicalOrdering` returns a permutation that sorts them this way, and thus makes the graph's adjacency matrix upper triangular.

In[332]:=

```
perm = IGTopologicalOrdering[ag]
```

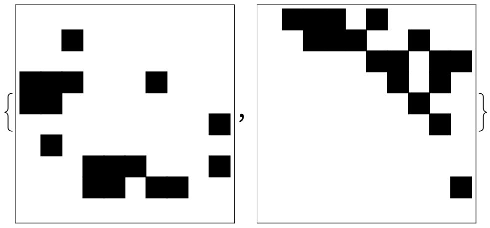
Out[332]=

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

In[333]:=

```
With[{am = AdjacencyMatrix[ag]},
 ArrayPlot /@ {am, am⟦perm, perm⟧}
]
```

Out[333]=



### References

- P. Eades, X. Lin, and W. F. Smyth, A fast and effective heuristic for the feedback arc set problem, *Inf. Process. Lett.* **47**, 319 (1993). https://doi.org/10.1016/0020-0190(93)90079-O

## Chordal graphs

Chordal graphs are graphs that do not contain induced cycles with more than three vertices.

### IGChordalQ

In[334]:=

```
? IGChordalQ
```

IGChordalQ[graph] tests if graph is chordal.

A graph is chordal if each of its cycles of four or more nodes has a chord, i.e. an edge joining two non-adjacent vertices in the cycle. Equivalently, all chordless cycles in a chordal graph have at most 3 vertices.

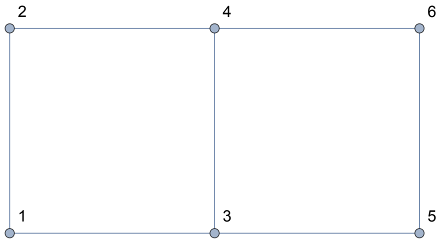
Chordal graphs are also called *rigid circuit graphs* or *triangulated graphs*.

Grid graphs are not chordal because they have chordless 4 cycles.

In[335]:=

```
g = GridGraph[{2, 3}, VertexLabels → "Name"]
```

Out[335]=




In[336]:=

```
IGChordalQ[g]
```

Out[336]=

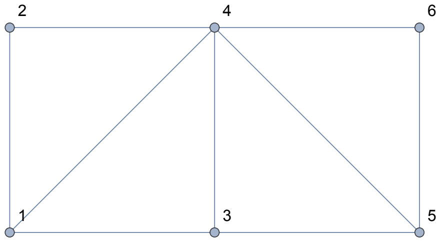
```
False
```

Adding chords to the 4 cycles makes them chordal.

In[337]:=

```
EdgeAdd[g, {1 ⟷ 4, 4 ⟷ 5}]
```

Out[337]=




In[338]:=

```
IGChordalQ[%]
```

Out[338]=

```
True
```

## IGChordalCompletion

In[339]:=

```
? IGChordalCompletion
```

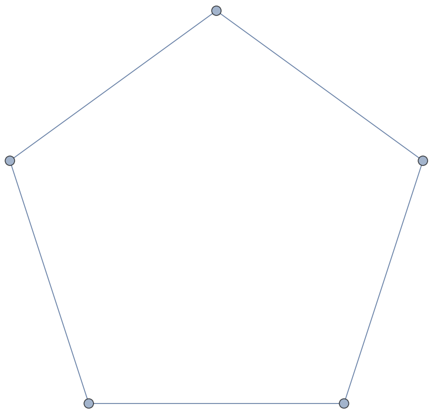
IGChordalCompletion[graph] gives a set of edges that, when added to graph, make it chordal. The edge–set this function returns is usually not minimal.

`IGChordalCompletion` computes a set of edges that, when added to a graph, make it chordal. The edge set returned is not usually minimal, i.e. some of the edges may not be necessary to create a chordal graph.

In[340]:=

```
g = CycleGraph[5]
```

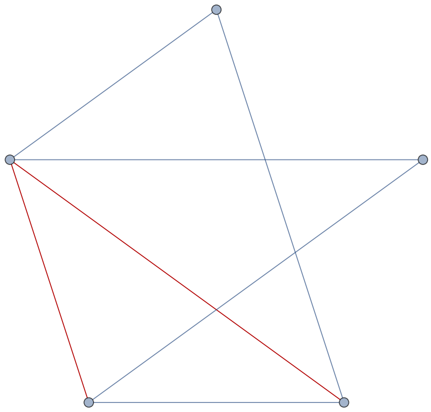
Out[340]=



In[341]:=

```
completion = IGChordalCompletion[g];
HighlightGraph[EdgeAdd[g, completion], completion]
```

Out[342]=



## IGMaximumCardinalitySearch

In[343]:=

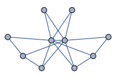
```
? IGMaximumCardinalitySearch
```

IGMaximumCardinalitySearch[graph] assigns a rank to each vertex, from 1 to n, according to the maximum cardinality search algorithm. Visiting the vertices of the graph by decreasing rank is equivalent to always visiting the next vertex with the most already visited neighbours.

The maximum cardinality search algorithm visits the vertices of the graph in such an order so that every time the vertex with the most already visited neighbours is visited next. Ties are broken arbitrarily. Then vertices are assigned ranks $\alpha$ in decreasing order from the vertex count of the graph to 1. `IGMaximumCardinalitySearch` returns these ranks.

The visiting order is animated below:

In[344]:=

g =  ;

In[345]:=

```
ranks = AssociationThread[VertexList[g], IGMaximumCardinalitySearch[g]]
```

Out[345]=

```
<|1 → 10, 2 → 3, 3 → 8, 4 → 6, 5 → 7, 6 → 2, 7 → 1, 8 → 5, 9 → 4, 10 → 9|>
```

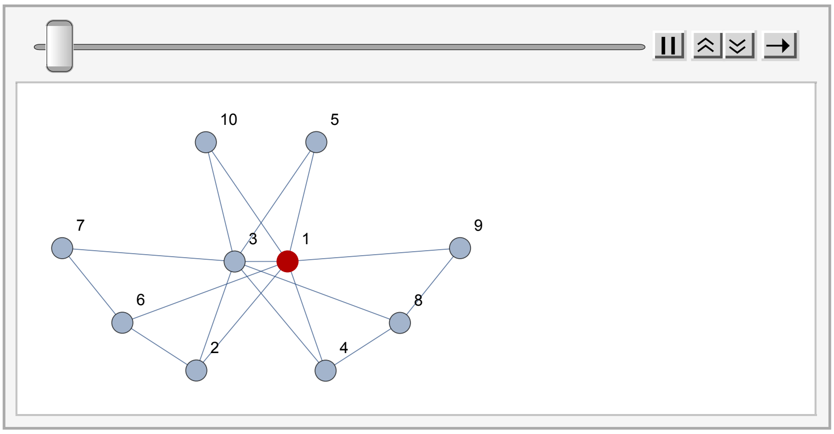
In[346]:=

```
verts = Keys@Reverse@Sort[ranks]
Table[
  HighlightGraph[
   Graph[g, VertexLabels → "Name"],
   Take[verts, i]
  ],
  {i, VertexCount[g]}
 ] // ListAnimate
```

Out[346]=

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

Out[347]=



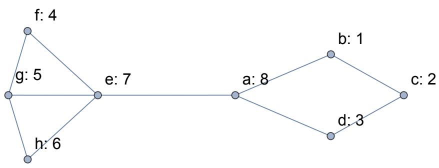

The rank $\alpha$ is useful for deciding the chordality of a graph. A graph is chordal if and only if any two neighbors of a vertex which are higher in rank than it are connected to each other.

Label the vertices of a graph with their ranks.

In[348]:=

```
g = IGShorthand["a-b-c-d-a-e-f-g-h-e-g"];
IGVertexMap[Row[{#1, ": ", #2}] &, VertexLabels → {VertexList, IGMaximumCardinalitySearch}, g]
```

Out[349]=




Notice that vertex b has two higher-rank neighbours that are not connected to each other. This graph is not chordal. Use `IGChordalCompletion` to determine which edges to add to it to make it chordal.

In[350]:=

```
IGChordalCompletion[g]
```

Out[350]=

```
{c ⟷ a}
```

## References


- R. E. Tarjan, M. Yannakakis: Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs, SIAM J. Comput., 13(3), 566–579 (1984). https://doi.org/10.1137/0213035

## Clustering coefficient

In[351]:=

```
? IG*ClusteringCoefficient
```

IGraphM`

| | |
|---|---|
| IGAverageLocalClusteringCoefficient | IGLocalClusteringCoefficient |
| IGGlobalClusteringCoefficient | IGWeightedClusteringCoefficient |

Clustering coefficients are measures of the degree to which vertices in a graph tend to cluster together. They are also referred to as *transitivity*, as they measure how often two vertices that are connected through a third one are also directly connected.

All clustering coefficient calculations in IGraph/M ignore edge directions.

### IGGlobalClusteringCoefficient

In[352]:=

```
? IGGlobalClusteringCoefficient
```

IGGlobalClusteringCoefficient[graph] gives the global clustering coefficient of graph.

The clustering coefficient of an undirected graph is defined as

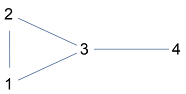
$$C = \frac{\text{number of closed ordered triplets}}{\text{number of connected ordered triplets}}$$

The available options are:

- `"ExcludeIsolates" → True` will cause `Indeterminate` to be returned if the graph has no connected triplets. With the default `"ExcludeIsolates" → False`, `0` is returned.

The following graph has 10 connected ordered triplets, namely {3, 1, 2}, {2, 1, 3}, {1, 2, 3}, {3, 2, 1}, {2, 3, 1}, {2, 3, 4}, {1, 3, 4}, {1, 3, 2}, {4, 3, 2}, {4, 3, 1}. Out of these, only 6 are closed: {1, 3, 2}, {1, 2, 3}, {2, 1, 3}, {2, 3, 1}, {3, 2, 1}, {3, 1, 2}. Thus the clustering coefficient is `6 / 10 = 0.6`.

In[353]:=

IGGlobalClusteringCoefficient[  ]



Out[353]=

```
0.6
```

### IGLocalClusteringCoefficient

In[354]:=

```
? IGLocalClusteringCoefficient
```

IGLocalClusteringCoefficient[graph] gives the local clustering coefficient of each vertex.

The local clustering coefficient of a vertex is defined as

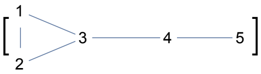
$$C = \frac{\text{number of connected pairs of neighbours}}{\text{total number of pairs of neighbours}}$$

The available options are:

- `"ExcludeIsolates" → True` will cause `Indeterminate` to be returned for degree 0 and degree 1 vertices. With the default `"ExcludeIsolates" → False`, `0` is returned.

The the following graph, vertex 4 has two neighbours which are disconnected, making its local clustering zero. However, vertex 5 has only one neighbour, thus computing the local clustering for it arguably does not make sense. Setting `"ExcludeIsolates" → True` serves to distinguish these two cases by returning `Indeterminate` for vertex 5.

In[355]:=

`IGLocalClusteringCoefficient[` 

`]`

Out[355]=

```
{1., 1., 0.333333, 0., 0.}
```

In[356]:=

`IGLocalClusteringCoefficient[` 

`, "ExcludeIsolates" → True]`

Out[356]=

```
{1., 1., 0.333333, 0., Indeterminate}
```

## IGAverageLocalClusteringCoefficient

In[357]:=

```
? IGAverageLocalClusteringCoefficient
```

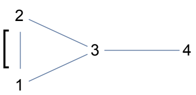
IGAverageLocalClusteringCoefficient[graph] gives the average local clustering coefficient of graph.

The available options are:

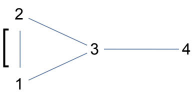
- `"ExcludeIsolates" → True` will cause degree 0 and degree 1 vertices to be excluded from the calculation.

With `"ExcludeIsolates" → True`, the local clustering coefficient of vertex 4 will be excluded from the calculation of the average.

In[358]:=

`{IGAverageLocalClusteringCoefficient[` 

`],`

`IGAverageLocalClusteringCoefficient[` 

`, "ExcludeIsolates" → True]}`

Out[358]=

```
{0.583333, 0.777778}
```

When the graph has no vertices with degree of at least 2, and `"ExcludeIsolates" → True` is set, the result will be `Indeterminate`.

In[359]:=

`IGAverageLocalClusteringCoefficient[`  `, "ExcludeIsolates" → True]`

Out[359]=

```
Indeterminate
```

## IGWeightedClusteringCoefficient

In[360]:=

```
? IGWeightedClusteringCoefficient
```

IGWeightedClusteringCoefficient[graph] gives the weighted local clustering coefficient, as defined by A. Barrat et al. (2004) http://dx.doi.org/10.1073/pnas.0400087101

`IGWeightedClusteringCoefficient` computes the *weighted* local clustering coefficient. This function expects a weighted graph as input.

The available options are:

- `"ExcludeIsolates" → True` will cause `Indeterminate` to be returned for degree 0 and degree 1 vertices. With the default `"ExcludeIsolates" → False`, `0` is returned.

### References


- A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, The architecture of complex weighted networks, PNAS 101, 3747 (2004). https://dx.doi.org/10.1073/pnas.0400087101


## Neighbour degrees

### IGAverageNeighborDegree

In[361]:=

```
? IGAverageNeighborDegree
```

IGAverageNeighborDegree[graph] gives the average neighbour degree of the vertices of graph.
IGAverageNeighborDegree[graph, {vertex1, vertex2, …}] gives the average neighbour degree of the specified vertices.
IGAverageNeighborDegree[graph, All, mode] uses the given mode, "In", "Out" or "All", to find neighbours and degrees in directed graphs. The default is "Out".
IGAverageNeighborDegree[graph, All, degreeMode, neighborMode] uses different modes for finding neighbours and degrees.

`IGAverageNeighborDegree` computes the average of the degrees of each vertex's neighbours. In weighted graphs, a weighted average is used:

$$k_{\mathrm{nn},u} = \frac{1}{s_u} \sum_v w_{uv}\, k_v$$

$k_{\mathrm{nn},u}$ denotes the average neighbour degree of vertex $u$, $k_v$ is the degree of vertex $v$, $w_{uv}$ is the weighted adjacency matrix, and $s_u = \sum_v w_{uv}$ is the strength of vertex $u$.

`IGAverageNeighborDegree` is similar to `MeanNeighborDegree`, with a few differences: it can compute the measure for only a subset of vertices, the interpretation of degrees and neighbours can be controlled independently in directed graphs, and for vertices which have no neighbours it returns `Indeterminate` instead of `0`.

Average neighbour degree in a star graph:

In[362]:=

```
IGAverageNeighborDegree[StarGraph[4]]
```

Out[362]=

```
{1., 3., 3., 3.}
```

Compute the result only for vertices 1 and 3:

In[363]:=

```
IGAverageNeighborDegree[StarGraph[4], {1, 3}]
```

Out[363]=

```
{1., 3.}
```

`All` computes the result for all vertices (the default):

In[364]:=

```
IGAverageNeighborDegree[StarGraph[4], All]
```

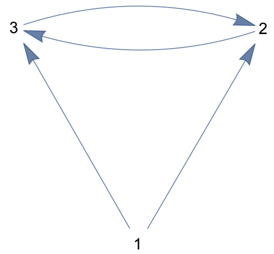
Out[364]=

```
{1., 3., 3., 3.}
```

When a vertex has no neighbours, `Indeterminate` is returned:

In[365]:=
```
IGAverageNeighborDegree[IGShorthand["1,2-3"]]
```
Out[365]=
```
{Indeterminate, 1., 1.}
```

In directed graphs, the out-degrees of out-neighbours are considered by default.

In[366]:=
```
g = 
```


```
;
```

In[367]:=
```
IGAverageNeighborDegree[g]
```
Out[367]=
```
{1., 1., 1.}
```

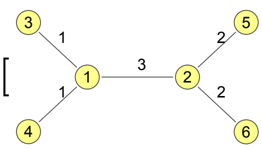
Use in-degrees of in-neighbours instead:

In[368]:=
```
IGAverageNeighborDegree[g, All, "In"]
```
Out[368]=
```
{Indeterminate, 1., 1.}
```

Use the in-degrees of all neighbours:

In[369]:=
```
IGAverageNeighborDegree[g, All, "In", "All"]
```
Out[369]=
```
{2., 1.33333, 1.33333}
```

Compute a weighted neighbour degree average. The weights used in averaging are taken from the edge weights:

In[370]:=
```
IGAverageNeighborDegree[
```


```
]
```
Out[370]=
```
{2.2, 1.85714, 3., 3., 3., 3.}
```

### References


- A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, The architecture of complex weighted networks, PNAS 101, 3747 (2004). https://dx.doi.org/10.1073/pnas.0400087101

## IGAverageDegreeConnectivity

In[371]:=
```
? IGAverageDegreeConnectivity
```

IGAverageDegreeConnectivity[graph] gives the average neighbour degree for vertices of degree k=1, 2, …
IGAverageDegreeConnectivity[graph, mode] uses the given mode, "In", "Out" or "All", to find neighbours and degrees in directed graphs. The default is "Out".
IGAverageDegreeConnectivity[graph, degreeMode, neighborMode] uses different modes for finding neighbours and degrees.

`IGAverageDegreeConnectivity` computes the average neighbour degree as a function of the vertex degree. The *i*th element of the result is the average of the `IGAverageNeighborDegree` result for all vertices of degree *i*.

In[372]:=
```
g = RandomGraph[{30, 50}];
```

In[373]:=
```
IGAverageDegreeConnectivity[g]
```

Out[373]=
```
{3., 3.92857, 3.44444, 4.21429, 4.46667, 4.33333, Indeterminate, 4.125}
```

An equivalent implementation of `IGAverageDegreeConnectivity` is:

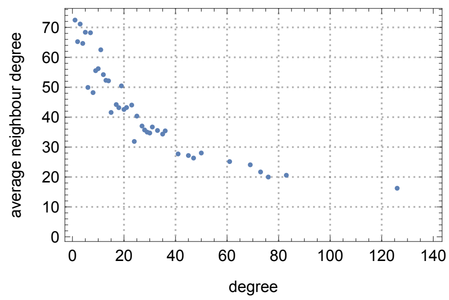
In[374]:=
```
Transpose[{VertexDegree[g], IGAverageNeighborDegree[g]}] //
  GroupBy[#, First → Last, Mean] & //
 Lookup[#, Range@Max@VertexDegree[g], Indeterminate] &
```

Out[374]=
```
{3., 3.92857, 3.44444, 4.21429, 4.46667, 4.33333, Indeterminate, 4.125}
```

Compute the average degree connectivity curve for a scale free network:

In[375]:=
```
ListPlot[
 IGAverageDegreeConnectivity@IGStaticPowerLawGame[1000, 2000, 2],
 FrameLabel → {"degree", "average neighbour degree"},
 PlotTheme → "Detailed"
]
```

Out[375]=




### References


- A. Barrat, M. Barthélemy, R. Pastor-Satorras, and A. Vespignani, The architecture of complex weighted networks, PNAS 101, 3747 (2004). https://dx.doi.org/10.1073/pnas.0400087101


## Shortest paths

The length of a path between two vertices is the number of edges the path consists of. Functions that use edge weights consider the path length to be the sum of edge weights along the path.

## IGDistanceMatrix

In[376]:=

```
? IGDistanceMatrix
```

IGDistanceMatrix[graph] gives the shortest path length between each vertex pair
in graph. Available Method options: {"Unweighted", "Dijkstra", "BellmanFord", "Johnson"}.
IGDistanceMatrix[graph, fromVertices] gives the shortest path lengths between from the given vertices to each vertex in graph.
IGDistanceMatrix[graph, fromVertices, toVertices] gives the shortest path lengths between the given vertices in graph.

`IGDistanceMatrix` takes the following `Method` options:

- `Automatic` selects a method automatically. As of IGraph/M 0.5, `"Unweighted"` is selected for unweighted graphs, `"Dijkstra"` for weighted graphs with only positive weights, and `"Johnson"` otherwise.
- `"Unweighted"` ignores weights
- `"Dijkstra"` uses Dijkstra's algorithm. All weights must be non-negative.
- `"BellmanFord"` uses the Bellman–Ford algorithm. Negative weights are supported but all cycles must have a non-negative total weight.
- `"Johnson"` uses the Johnson algorithm. Negative weights are supported but all cycles must have a non-negative total weight.

The igraph C core may override explicit method settings when appropriate. For example, if the graph is not weighted, it always uses `"Unweighted"`.

## IGDistanceCounts

In[377]:=

```
? IGDistanceCounts
```

IGDistanceCounts[graph] gives a histogram of unweighted shortest path
lengths between all vertex pairs. The kth element of the result is the count of shortest paths
of length k. In undirected graphs, each path is counted only along one traversal direction.
IGDistanceCounts[graph, fromVertices] gives a histogram
of unweighted shortest path lengths from the given vertices to all others.

`IGDistanceCounts[graph]` counts all-pair *unweighted* shortest path lengths in the graph.
`IGDistanceCounts[graph, {v1, v2, …}]` counts unweighted shortest path lengths for paths starting at the given vertices.

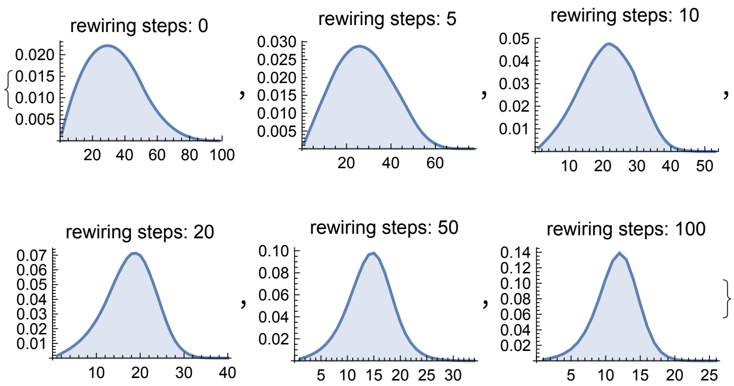
For weighted path lengths, or to restrict the computation to both certain start and end vertex sets, use `IGDistanceHistogram[]`.

Compute how the shortest path length distribution changes as we rewire a grid graph k times.

In[378]:=

```
Table[
 ListPlot[
  Normalize[IGDistanceCounts@IGRewire[GridGraph[{50, 50}], k], Total],
  Joined → True, Filling → Bottom, PlotLabel → StringTemplate["rewiring steps: ``"][k]
 ],
 {k, {0, 5, 10, 20, 50, 100}}
]
```

Out[378]=




## IGNeighborhoodSize

In[379]:=

```
? IGNeighborhoodSize
```

IGNeighborhoodSize[graph, vertex] gives the number of direct neighbours of vertex, i.e. its degree.
IGNeighborhoodSize[graph, All] gives the number of direct neighbours of all vertices.
IGNeighborhoodSize[graph, {vertex1, vertex2, …}] gives the number of direct neighbours of the specified vertices.
IGNeighborhoodSize[graph, All, max] gives the number of vertices reachable in at most max hops.
IGNeighborhoodSize[graph, All, {n}] gives the number of vertices reachable in precisely n hops.
IGNeighborhoodSize[graph, All, {min, max}] gives
  the number of vertices reachable in between min and max hops (inclusive).
IGNeighborhoodSize[graph, All, {min, max}, mode] uses
  the given mode, "In", "Out" or "All", when finding neighbours in directed graphs.

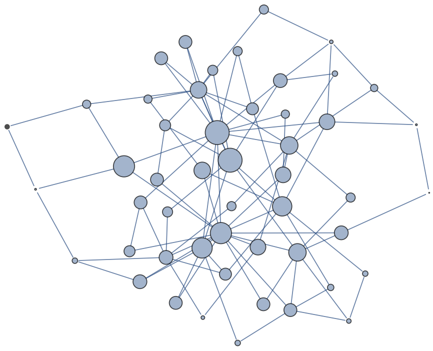
`IGNeighborhoodSize` returns the number of vertices reachable within a certain distance range from a given vertex, or from multiple given vertices.

Scale vertices proportionally to the number of their second order neighbours:

In[380]:=

```
g = IGBarabasiAlbertGame[50, 2, DirectedEdges → False];
IGVertexMap[# &, VertexSize → (Rescale@IGNeighborhoodSize[#, All, {2}] &), g]
```

Out[381]=



## IGDistanceHistogram

In[382]:=

```
? IGDistanceHistogram
```

IGDistanceHistogram[graph, binsize] gives a histogram of weighted all–pair shortest path lengths in graph with the given bin size. In the case of undirected graphs, path lengths are double counted. Available Method options: {"Dijkstra", "BellmanFord"}.

IGDistanceHistogram[graph, binsize, from] gives a histogram of weighted shortest path lengths in graph for the given starting vertices and bin size.

IGDistanceHistogram[graph, binsize, from, to] gives a histogram of weighted shortest path lengths in graph for the given starting and ending vertices and bin size.

`IGDistanceHistogram[]` computes the weighted shortest path length histogram between the specified start and end vertex sets. The start and end vertex sets do not need to be the same. Note that if the graph is undirected, path lengths between `s` and `t` will be double counted (from `s` → `t` and `t` → `s`) if `s` and `t` appear both in the starting and ending vertex sets.

`IGDistanceHistogram[]` is useful when the result of `IGDistanceMatrix[]` (or `GraphDistanceMatrix[]`) does not fit in memory.

## IGAveragePathLength

In[383]:=

```
? IGAveragePathLength
```

IGAveragePathLength[graph] returns the average of all–pair shortest path lengths of the graph. Vertex pairs between which there is no path are excluded. Available Method options: {"Unweighted", "Dijkstra", "BellmanFord", "Johnson"}.

`IGAveragePathLength` computes the average pairwise distances between vertices.

Available options:

- `Method` can take the values `"Unweighted"`, `"Dijkstra"`, `"BellmanFord"`, `"Johnson"` and `Automatic`. `Automatic` uses `"Unweighted"` if no edge weights are present, `"Dijkstra"` if all weights are non-negative and `"Johnson"` otherwise.

- "ByComponents" controls how unconnected graphs are handled. If False, Infinity is returned. If True, vertex pairs between which there is no path are excluded from the calculation.

## IGGirth

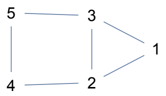
In[384]:=

```
? IGGirth
```

IGGirth[graph] returns the length of the shortest cycle of the graph. The graph is treated as undirected, self–loops and multi–edges are ignored.

IGGirth computes the girth of a graph, i.e. the length of its shortest cycle. IGGirth ignores multi-edges and self-loops. Edge directions and edge weights are also ignored.

In[385]:=

IGGirth[  ]

Out[385]=

3

If the graph has no cycles, ∞ is returned.

In[386]:=

```
IGGirth@IGShorthand["1-2"]
```

Out[386]=

∞

## IGDiameter and IGFindDiameter

In[387]:=

```
? IGDiameter
```

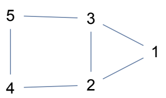
IGDiameter[graph] gives the diameter of graph. Available Method options: {"Unweighted", "Dijkstra"}.

The diameter of a graph is the length of the longest shortest path between any two vertices.

The available options are:

- Method can take the values "Unweighted", "Dijkstra" or Automatic. "Dijkstra" takes edge weights into account. Automatic chooses based on whether the graph is weighted.
- "ByComponents" controls how unconnected graphs are handled. If False, Infinity is returned. If True, the longest shortest path is returned. In the undirected case, this is the largest diameter of any connected component.

In[388]:=

IGDiameter[  ]

Out[388]=

2

For the null graph, Indeterminate is returned.

In[389]:=

```
IGDiameter[IGEmptyGraph[]]
```

Out[389]=

Indeterminate

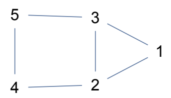
In[390]:=

```
? IGFindDiameter
```

IGFindDiameter[graph] returns a longest shortest path in graph, i.e. a shortest path with length equal to the graph diameter. Available Method options: {"Unweighted", "Dijkstra"}.

In[391]:=

```
IGFindDiameter[
```




```
]
```

Out[391]=

```
{1, 2, 4}
```

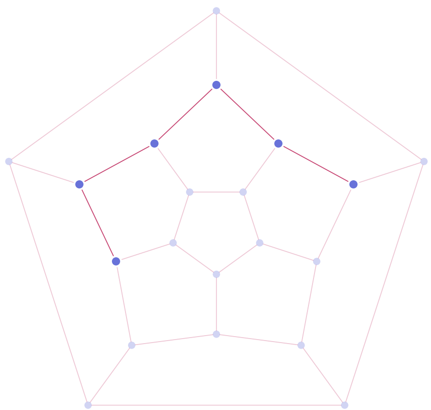
In[392]:=

```
g = dodecahedral graph GRAPH ["Graph"];
```

In[393]:=

```
HighlightGraph[g, PathGraph@IGFindDiameter[g],
 GraphHighlightStyle → "DehighlightFade", PlotTheme → "RoyalColor"
]
```

Out[393]=



## IGEccentricity

In[394]:=

```
? IGEccentricity
```

IGEccentricity[graph] returns the eccentricity of all vertices.
IGEccentricity[graph, vertex] returns the eccentricity of the given vertex.
IGEccentricity[graph, {vertex1, vertex2, …}] returns the eccentricity of the given vertices.

The eccentricity of a vertex is the longest shortest path to any other vertex. `IGEccentricity` computes the *unweighted* eccentricity of each vertex within the connected component where it is contained.

In[395]:=

```
IGEccentricity@CycleGraph[8]
```

Out[395]=

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

Connected components are considered separately.

In[396]:=

```
IGEccentricity[IGDisjointUnion[{CycleGraph[3], CycleGraph[8]}]]
```

Out[396]=

```
{1, 1, 1, 4, 4, 4, 4, 4, 4, 4, 4}
```

## IGRadius

In[397]:=

```
? IGRadius
```

IGRadius[graph] returns the unweighted graph radius.

The radius of a graph is the smallest eccentricity of any of its vertices, i.e. the eccentricity of the graph center.

## IGVoronoiCells

In[398]:=

```
? IGVoronoiCells
```

IGVoronoiCells[graph, {v1, v2, …}] returns the sets of vertices closest to each given vertex.

`IGVoronoiCells[graph, centers]` partitions a graph's vertices into groups based on which given centre vertex they are the closest to. Edge weights are considered for the distance calculations.

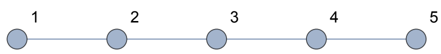
Available options:

- `"Tiebreaker"` sets the function used to decide which cell a vertex should belong to if its distance to several different centres is equal. The default is to use the first qualifying cell. Possible useful settings are `First`, `Last`, `RandomChoice`.

In[399]:=

```
g = PathGraph[Range[5], VertexLabels → "Name", VertexSize → Medium]
```

Out[399]=




In[400]:=

```
IGVoronoiCells[g, {2, 4}]
```

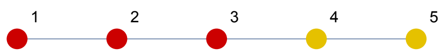
Out[400]=

```
<|2 → {1, 2, 3}, 4 → {4, 5}|>
```

In[401]:=

```
HighlightGraph[g, Values[%]]
```

Out[401]=




In the event of a tie, a vertex is added to the first qualifying cell. The tiebreaker function can be changed as below.

In[402]:=

```
IGVoronoiCells[g, {2, 4}, "Tiebreaker" → Last]
```

Out[402]=

```
<|2 → {1, 2}, 4 → {3, 4, 5}|>
```

In[403]:=

```
Table[IGVoronoiCells[g, {2, 4}, "Tiebreaker" → RandomChoice], {5}]
```

Out[403]=

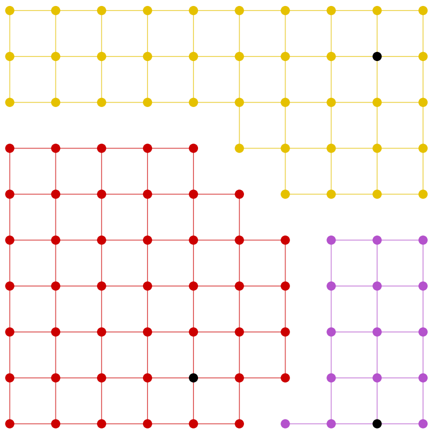
```
{<|2 → {1, 2, 3}, 4 → {4, 5}|>, <|2 → {1, 2}, 4 → {3, 4, 5}|>,
 <|2 → {1, 2, 3}, 4 → {4, 5}|>, <|2 → {1, 2, 3}, 4 → {4, 5}|>, <|2 → {1, 2}, 4 → {3, 4, 5}|>}
```

Find Voronoi cells on a grid.

In[404]:=

```
g = GridGraph[{10, 10}, VertexSize → Medium, GraphStyle → "BasicBlack"];
centers = RandomSample[VertexList[g], 3];
HighlightGraph[g,
 Append[
  Subgraph[g, #] & /@ Values@IGVoronoiCells[g, centers],
  Style[centers, Black]
 ],
 GraphHighlightStyle → "DehighlightHide"
]
```

Out[406]=

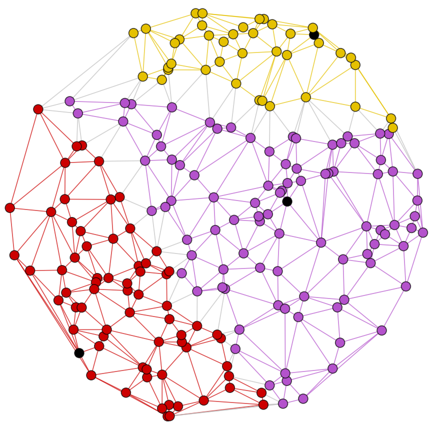


Edge weights are interpreted as distances.

In[407]:=

```
g = IGMeshGraph@DelaunayMesh@RandomPoint[Disk[], 200];
centers = RandomSample[VertexList[g], 3];
HighlightGraph[g,
 Append[
  Subgraph[g, #] & /@ Values@IGVoronoiCells[g, centers],
  Style[centers, Black]
 ],
 GraphHighlightStyle → "DehighlightGray"
]
```

Out[409]=



### IGShortestPathTree

In[410]:=

```
? IGShortestPathTree
```

IGShortestPathTree [graph, vertex] give the shortest path tree of graph rooted in vertex.

**Experimental:** This is experimental functionality that may change in the future.

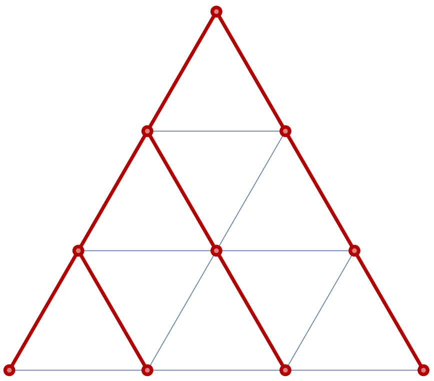
In[411]:=

```
g = IGTriangularLattice[4];
```

In[412]:=

```
HighlightGraph[g, IGShortestPathTree[g, 1], GraphHighlightStyle → "Thick"]
```

Out[412]=



## Efficiency measures

### IGGlobalEfficiency

In[413]:=

```
? IGGlobalEfficiency
```

IGGlobalEfficiency[graph] gives the global efficiency of graph.

`IGGlobalEfficiency[graph]` computes the global efficiency of a graph. The global efficiency is defined as the average inverse shortest path length between all pairs of vertices,

$$E_{\text{global}} = \frac{1}{V(V-1)} \sum_{u,v} \frac{1}{d_{uv}},$$

where $d_{uv}$ is the graph distance from vertex $u$ to vertex $v$ and $V$ is the number of vertices. When $v$ is not reachable from $u$, $1/d_{uv}$ is taken to be 0.

Available options:

- `DirectedEdges → False` ignores edge directions when computing shortest path lengths.

Compute the global efficiency of a network …

In[414]:=

```
g = ExampleData[{"NetworkGraph", "ProteinInteraction"}];
IGGlobalEfficiency[g]
```

Out[415]=

```
0.0997348
```

… and that of its spanning tree.

In[416]:=

```
IGGlobalEfficiency@IGSpanningTree[g]
```

Out[416]=

```
0.00106384
```

### References

- V. Latora and M. Marchiori, Efficient behavior of small-world networks, Phys. Rev. Lett. 87, 198701 (2001). https://dx.doi.org/10.1103/PhysRevLett.87.198701

## IGLocalEfficiency

In[417]:=

```
? IGLocalEfficiency
```

IGLocalEfficiency[graph] gives the local efficiency around each vertex of graph.
IGLocalEfficiency[graph, {vertex1, vertex2, …}] gives the local efficiency around the given vertices.
IGLocalEfficiency[graph, All, "Out"] uses outgoing edges to define the neighbourhood in a directed graph.

`IGLocalEfficiency[graph]` computes the local efficiency around each vertex of a graph. The local efficiency around a vertex $u$ is defined as the average pairwise inverse shortest path length between the neighbours of $u$ after excluding $u$ itself from the graph,

$$E_{\text{local}}(u) = \frac{1}{k_u(k_u - 1)} \sum_{v,w \in N(u)} \frac{1}{d_{vw}},$$

where $k_u$ is the degree of vertex $u$, $N(u)$ denotes its neighbourhood and $d_{vw}$ is the graph distance from vertex $v$ to vertex $w$. If $u$ has less than two neighbours, $E_{\text{local}}(u)$ is taken to be 0.

Available options:

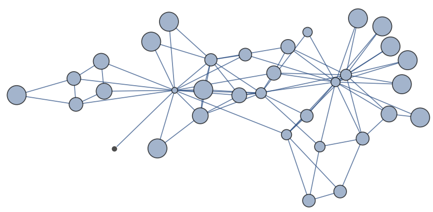
- `DirectedEdges → False` ignores edge directions when computing shortest path lengths.

Size the vertices of a graph according to the corresponding local efficiency

In[418]:=

```
g = ExampleData[{"NetworkGraph", "ZacharyKarateClub"}];
IGVertexMap[1.5 # &, VertexSize → IGLocalEfficiency, g]
```

Out[419]=

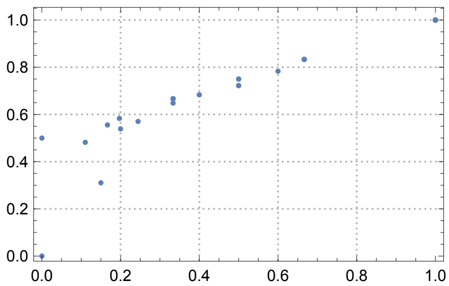


Plot the local efficiency versus the local clustering coefficient.

In[420]:=

```
ListPlot[
 Transpose[{IGLocalClusteringCoefficient[g], IGLocalEfficiency[g]}],
 PlotTheme → "Detailed"
]
```

Out[420]=




Compute the local efficiency of a subset of vertices only.

In[421]:=

```
g = RandomGraph[{10, 20}, DirectedEdges → True];
IGLocalEfficiency[g, {1, 2, 3}]
```

Out[422]=

```
{0.506944, 0.408333, 0.5}
```

By default, both in- and out-neighbours are considered when determining the neighbourhoods of vertices. We can also consider only in-neighbours or only out-neighbours.

In[423]:=

```
{IGLocalEfficiency[g, All, "All"],
 IGLocalEfficiency[g, All, "In"],
 IGLocalEfficiency[g, All, "Out"]}
```

Out[423]=

```
{{0.506944, 0.408333, 0.5, 0.388889, 0.275, 0.305556, 0.375, 0.336111, 0.493333, 0.35},
 {1., 0.625, 0.5, 0.5, 0.25, 0.25, 0.375, 0.1, 0.416667, 0.},
 {0.125, 0., 0., 0.25, 0.388889, 0.25, 0., 0.5, 0.520833, 0.35}}
```

Ignore edge directions when computing shortest paths.

In[424]:=

```
IGLocalEfficiency[g, DirectedEdges → False]
```

Out[424]=

```
{0.722222, 0.833333, 1., 0.75, 0.561667, 0.5, 0.583333, 0.583333, 0.7, 0.5}
```

### References


- I. Vragović, E. Louis, and A. Díaz-Guilera, Efficiency of informational transfer in regular and complex networks, Phys. Rev. E 71, 1 (2005). https://dx.doi.org/10.1103/PhysRevE.71.036122


## IGAverageLocalEfficiency

In[425]:=

```
? IGAverageLocalEfficiency
```

IGAverageLocalEfficiency[graph] gives the average local efficiency of graph.
IGAverageLocalEfficiency[graph, "Out"] uses outgoing edges to define the neighbourhood in a directed graph.

`IGAverageLocalEfficiency[graph]` computes the average local efficiency of a network. See

`IGLocalEfficiency` for a definition of this graph measure.

Plot the decrease in average local efficiency during sequential edge removals.

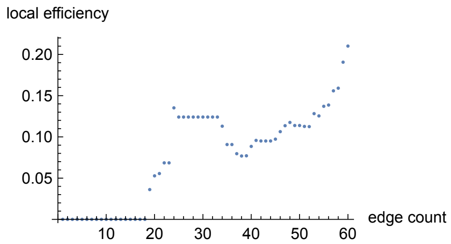
In[426]:=

```
g = RandomGraph[{30, 60}, DirectedEdges → True];
```

In[427]:=

```
ListPlot[
 Table[
  {k, IGAverageLocalEfficiency@Graph[VertexList[g], Take[EdgeList[g], k]]},
  {k, EdgeCount[g]}
 ],
 AxesLabel → {"edge count", "local efficiency"}
]
```

Out[427]=




`IGAverageLocalEfficiency` simply gives the average of the values returned by `IGLocalEfficiency`.

In[428]:=

```
{IGAverageLocalEfficiency[g], Mean@IGLocalEfficiency[g]}
```

Out[428]=

```
{0.210095, 0.210095}
```

Use only the out-neighbourhood while computing the local efficiency.

In[429]:=

```
IGAverageLocalEfficiency[g, "Out"]
```

Out[429]=

```
0.142476
```

## Bipartite graphs

The vertices of a bipartite graph can be divided into two groups (partitions) such that connections run only between the two partitions, but never within a single partition.

In[430]:=

```
? IGBipartite*
```

▼ IGraphM`

| | | |
|---|---|---|
| IGBipartiteGameGNM | IGBipartiteIncidenceMatrix | IGBipartiteQ |
| IGBipartiteGameGNP | IGBipartitePartitions | |
| IGBipartiteIncidenceGraph | IGBipartiteProjections | |

## IGBipartiteQ

In[431]:=

```
? IGBipartiteQ
```

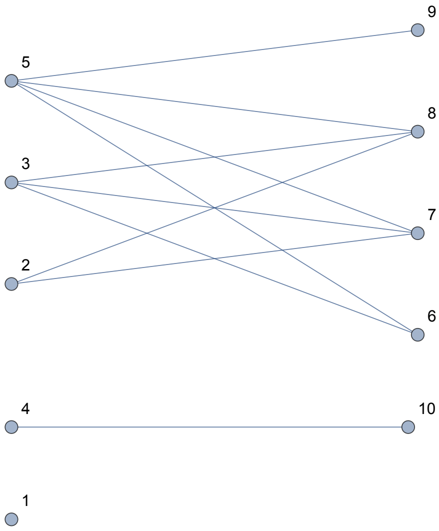
IGBipartiteQ[graph] tests if graph is bipartite.
IGBipartiteQ[graph, {vertices1, vertices2}] verifies that no edges are running between the two given vertex subsets.

Generate a graph and verify that it is bipartite.

In[432]:=

```
g = IGBipartiteGameGNM[5, 5, 10, VertexLabels → "Name"]
```

Out[432]=




In[433]:=

```
IGBipartiteQ[g]
```

Out[433]=

```
True
```

Verify that no edges run between two disjoint vertex subsets of the graph.

In[434]:=

```
IGBipartiteQ[g, {{1, 2, 3}, {6, 7, 8}}]
```

Out[434]=

```
True
```

## IGBipartitePartitions

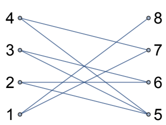
In[435]:=

```
? IGBipartitePartitions
```

IGBipartitePartitions[graph] partitions the vertices of a bipartite graph.
IGBipartitePartitions[graph, vertex] ensures that the first partition which is returned contains vertex.

Find a bipartite partitioning of a graph.

In[436]:=

g = 

;

In[437]:=

```
IGBipartitePartitions[g]
```

Out[437]=

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

Ensure that the partitions are returned in such an order that the first one contains vertex 5.

In[438]:=

```
IGBipartitePartitions[g, 5]
```

Out[438]=

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

`$Failed` is returned for non-bipartite graphs.

In[439]:=

```
IGBipartitePartitions[CompleteGraph[4]]
```

IGBipartitePartitions: The graph is not bipartite.

Out[439]=

```
$Failed
```

We can use `IGPartitionsToMembership` or `IGKVertexColoring[..., 2]` to obtain a partition index for each vertex.

In[440]:=

```
IGPartitionsToMembership[g]@IGBipartitePartitions[g]
```

Out[440]=

```
{1, 1, 1, 1, 2, 2, 2, 2}
```

In[441]:=

```
IGKVertexColoring[g, 2]
```

Out[441]=

```
{{1, 1, 1, 1, 2, 2, 2, 2}}
```

## IGBipartiteProjections

In[442]:=

```
? IGBipartiteProjections
```

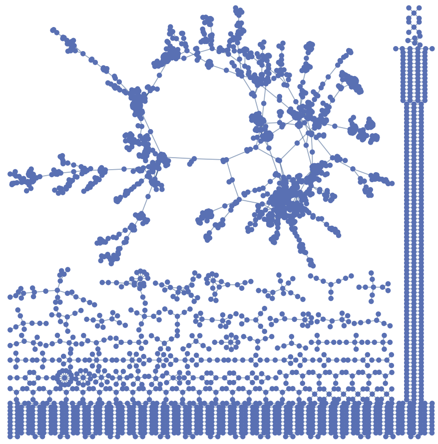
IGBipartiteProjections[graph] gives both bipartite projections of graph. Multiplicities are returned as edge weights. Edge directions are ignored.
IGBipartiteProjections[graph, {vertices1, vertices2}] returns both bipartite projections according to the specified partitioning.

The following bipartite graph described the relationship between diseases and genes.

In[443]:=

```
g = ExampleData[{"NetworkGraph", "BipartiteDiseasomeNetwork"}]
```

Out[443]=



In[444]:=

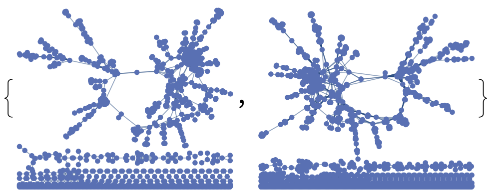
```
parts = Values@GroupBy[
    Thread[IGVertexProp["Type"][g] → VertexList[g]],
    First → Last
   ];
```

Construct a disease-disease and gene-gene network from it.

In[445]:=

```
IGBipartiteProjections[g, parts]
```

Out[445]=



## IGBipartiteIncidenceMatrix and IGBipartiteIncidenceGraph

In[446]:=

```
? IGBipartiteIncidenceGraph
```

IGBipartiteIncidenceGraph[mat] creates a bipartite graph from the given incidence matrix.
IGBipartiteIncidenceGraph[{vertices1, vertices2},
    mat] uses vertices1 and vertices2 as the vertex names in the two partitions.

In[447]:=

```
? IGBipartiteIncidenceMatrix
```

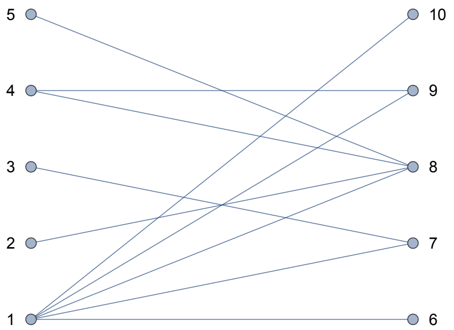
IGBipartiteIncidenceMatrix[graph] gives the incidence matrix of a bipartite graph.
IGBipartiteIncidenceMatrix[graph, {vertices1, vertices2}] uses the provided vertex partitioning.

Compute an incidence matrix. The default partitioning used by `IGBipartiteIncidenceMatrix` is the one returned by `IGBipartitePartitions`.

In[448]:=

```
g = IGBipartiteGameGNM[5, 5, 10, VertexLabels → "Name"]
```

Out[448]=




In[449]:=

```
bm = IGBipartiteIncidenceMatrix[g];
MatrixForm[bm, TableHeadings → IGBipartitePartitions[g]]
```

Out[450]//MatrixForm=

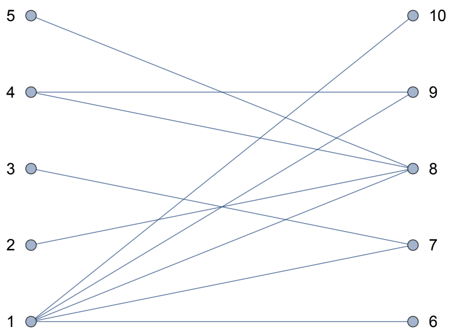
| | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 1 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 |

Reconstruct a graph from an incidence matrix.

In[451]:=

```
IGBipartiteIncidenceGraph[bm, VertexLabels → "Name", GraphLayout → "BipartiteEmbedding"]
```

Out[451]=




Compute an incidence matrix using a given partitioning / vertex ordering. It is allowed to pass only a subset of vertices.

In[452]:=

```
IGBipartiteIncidenceMatrix[g, {{1, 2, 3}, {6, 7, 8}}]
```

Out[452]=

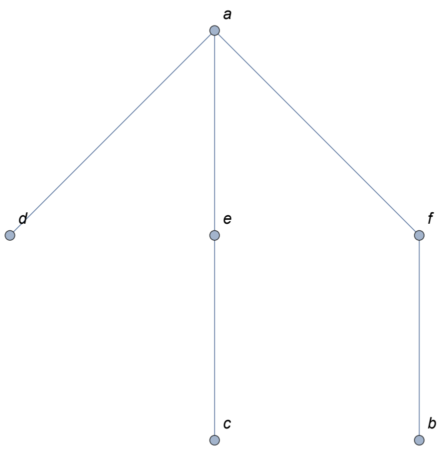
```
SparseArray[ Specified elements: 5
             Dimensions: {3, 3} ]
```

Reconstruct the bipartite graph while specifying vertex names.

In[453]:=

```
IGBipartiteIncidenceGraph[{{a, b, c}, {d, e, f}}, %, VertexLabels → "Name"]
```

Out[453]=




## Similarity measures

The functions in this section characterize the similarity of vertex pairs within a graph.

### IGBibliographicCoupling

In[454]:=

```
? IGBibliographicCoupling
```

IGBibliographicCoupling[graph] gives the bibliographic coupling between all vertex pairs in graph. The bibliographic coupling of two vertices is the number of vertices they both connect to (with directed edges).
IGBibliographicCoupling[graph, vertex] gives the bibliographic coupling of vertex with all other vertices in graph.
IGBibliographicCoupling[graph, {vertex1, vertex2, …}] gives the bibliographic coupling of vertex1, vertex2, … with all other vertices in graph.

The bibliographic coupling of two vertices in a directed graph is the number of other vertices they both connect to. The bibliographic coupling matrix can also be obtained using `am.am`$^{\mathsf{T}}$ – `DiagonalMatrix@VertexInDegree[g]`, where `am` is the adjacency matrix of the graph `g`.

In[455]:=

```
? IGStaticPowerLawGame
```

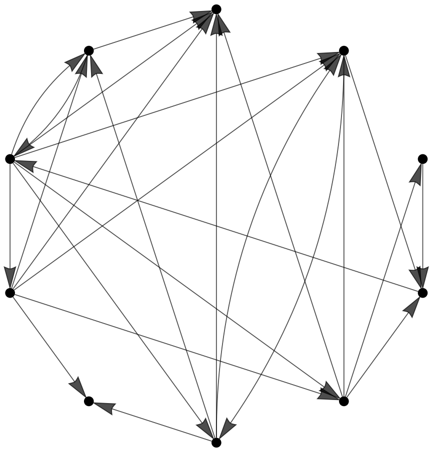
IGStaticPowerLawGame[n, m, exp] generates a random graph with n vertices and m edges, having a power–law degree distribution with the given exponent.
IGStaticPowerLawGame[n, m, expOut, expIn] generates a random directed graph with n vertices and m edges, having power–law in– and out–degree distributions with the given exponents.

Create a random graph and compute its bibliographic coupling matrix.

In[456]:=

```
g = IGStaticPowerLawGame[10, 25, 2, 4,
  GraphLayout → "CircularEmbedding", GraphStyle → "BasicBlack"]
```

Out[456]=



In[457]:=

```
cc = IGBibliographicCoupling[g];
MatrixForm[cc, TableHeadings → {VertexList[g], VertexList[g]}]
```

Out[458]//MatrixForm=

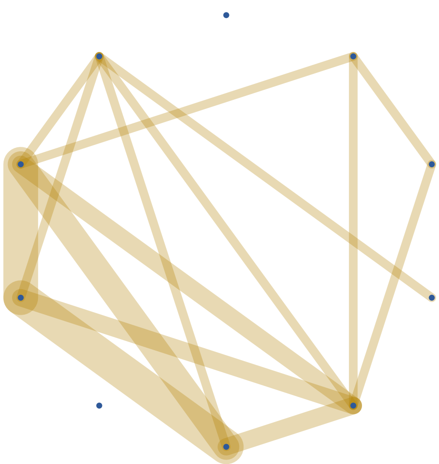
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 6 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 2 | 2 | 2 |
| 8 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 4 | 3 |
| 9 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 0 | 4 |
| 10 | 1 | 0 | 0 | 0 | 1 | 0 | 2 | 3 | 4 | 0 |

Construct the weighted graph corresponding to the bibliographic coupling of vertices and visualize it.

In[459]:=

```
ccg =
 IGWeightedAdjacencyGraph[cc, VertexCoordinates → GraphEmbedding[g], GraphStyle → "ThickEdge"] //
  IGEdgeMap[Thickness[0.02 #] &, EdgeStyle → IGEdgeProp[EdgeWeight]]
```

Out[459]=

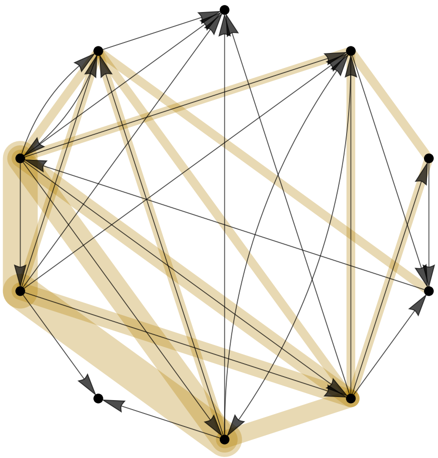


Overlay the bibliographic coupling graph with the original directed graph.

In[460]:=

```
Show[ccg, g]
```

Out[460]=



## IGCocitationCoupling

In[461]:=

```
? IGCocitationCoupling
```

IGCocitationCoupling[graph] gives the cocitation coupling between all vertex pairs in graph. The cocitation coupling of two vertices is the number of vertices connecting to both of them (with directed edges).
IGCocitationCoupling[graph, vertex] gives the cocitation coupling of vertex with all other vertices in graph.
IGCocitationCoupling[graph, {vertex1, vertex2, …}] gives the cocitation coupling of vertex1, vertex2, … with all other vertices in graph.

The co-citation coupling of two vertices in a directed graph is the number of other vertices that connect to both of them. The co-citation coupling matrix can also be obtained using `am`$^\mathsf{T}$`.am - DiagonalMatrix@VertexOutDegree[g]`, where am is the adjacency matrix of the graph g.

## IGDiceSimilarity

In[462]:=

```
? IGDiceSimilarity
```

IGDiceSimilarity[graph] gives the Dice similarity between all pairs of vertices.
IGDiceSimilarity[graph, {vertex1, vertex2, …}] gives the Dice similarity between the given vertices.

The Dice similarity coefficient of two vertices is twice the number of common neighbours divided by the sum of the degrees of the vertices. For directed graphs, out-neighbours are considered. Edge multiplicities are not taken into account.

The available options are:

- `SelfLoops → True` will include self-loops in the calculation of the similarity score.

## IGJaccardSimilarity

In[463]:=

```
? IGJaccardSimilarity
```

IGJaccardSimilarity[graph] gives the Jaccard similarity between all pairs of vertices.
IGJaccardSimilarity[graph, {vertex1, vertex2, …}] gives the Jaccard similarity between the given vertices.

The Jaccard similarity coefficient of two vertices is the number of common neighbours divided by the number of vertices that are neighbours of at least one of the two vertices being considered. For directed graphs, out-neighbours are considered. Edge multiplicities are not taken into account.

The available options are:

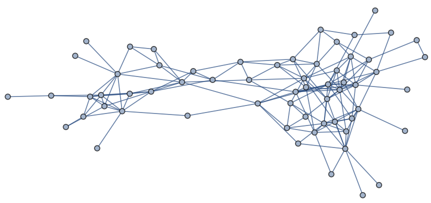
- `SelfLoops → True` will include self-loops in the calculation of the similarity score.

Construct and visualize a weighted graph of Jaccard similarities between vertices of an animal social network:

In[464]:=

```
g = ExampleData[{"NetworkGraph", "DolphinSocialNetwork"}]
```

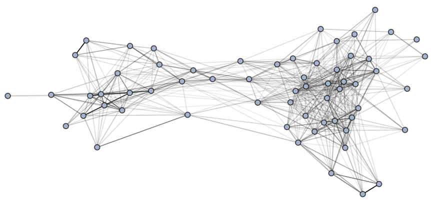
Out[464]=



In[465]:=

```
IGWeightedAdjacencyGraph[
  IGZeroDiagonal@IGJaccardSimilarity[g], VertexCoordinates → GraphEmbedding[g]] //
 IGEdgeMap[GrayLevel[0, #] &, EdgeStyle → IGEdgeProp[EdgeWeight]]
```

Out[465]=

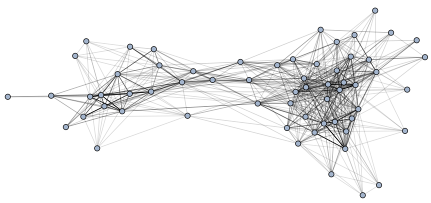


Compare it to the inverse log-weighted similarity:

In[466]:=

```
IGWeightedAdjacencyGraph[
  Rescale@IGInverseLogWeightedSimilarity[g], VertexCoordinates → GraphEmbedding[g]] //
 IGEdgeMap[GrayLevel[0, #] &, EdgeStyle → IGEdgeProp[EdgeWeight]]
```

Out[466]=



### IGInverseLogWeightedSimilarity

In[467]:=

```
? IGInverseLogWeightedSimilarity
```

IGInverseLogWeightedSimilarity[graph] gives the inverse log–weighted similarity between all pairs of vertices.
IGInverseLogWeightedSimilarity[graph, vertex] gives the inverse log–weighted similarity of vertex to all other vertices.
IGInverseLogWeightedSimilarity[graph, {vertex1, vertex2, …}] gives the inverse log–weighted similarity between the given vertices.

The inverse log-weighted similarity of two vertices is the number of their common neighbours, weighted by the inverse natural logarithm of the neighbours' degrees. It is also known as the Adamic–Adar index. It is based on the assumption that two vertices should be considered more similar if they share a low-degree common neighbour, since high-degree common neighbours are more likely to appear even by pure chance.

Formally, the similarity of vertices $u$ and $v$ is

$$A(u, v) = \sum_{w \in \mathcal{N}(u) \cap \mathcal{N}(v)} \frac{1}{\ln d_w},$$

where $\mathcal{N}(u)$ denotes the neighbourhood of vertex $u$ and $d_w$ denotes the degree of vertex $w$.

Isolated vertices will have zero similarity to any other vertex. Self-similarities are not calculated.

In directed graphs, the out-neighbours of each vertex are considered, weighted by the inverse logarithm of their in-degrees.

#### References


- Lada A. Adamic and Eytan Adar: Friends and neighbors on the Web, *Social Networks*, 25(3):211-230, 2003. https://doi.org/10.1016/S0378-8733(03)00009-1


## Connectivity and graph components

### IGConnectedQ and IGWeaklyConnectedQ

In[468]:=

```
? IGConnectedQ
```

IGConnectedQ[graph] tests if graph is strongly connected.

In[469]:=

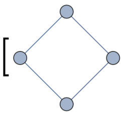
```
? IGWeaklyConnectedQ
```

IGWeaklyConnectedQ[graph] tests if graph is weakly connected.

`IGConnectedQ` checks if the graph is (strongly) connected. It is equivalent to `ConnectedGraphQ`. `IGWeaklyConnectedQ` check if a directed graph is weakly connected. It is equivalent to `WeaklyConnectedGraphQ`. Both of these functions use the implementation from the core igraph library, and will always be consistent with it for edge cases such as the null graph.

This graph is connected.

In[470]:=

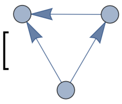
```
IGConnectedQ[
```

```
]
```

Out[470]=

```
True
```

This directed graph is only weakly connected.

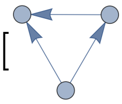
In[471]:=

```
IGConnectedQ[
```

```
]
```

Out[471]=

```
False
```

In[472]:=

```
IGWeaklyConnectedQ[
```

```
]
```

Out[472]=

```
True
```

The null graph is considered disconnected by convention.

In[473]:=

```
IGConnectedQ@IGEmptyGraph[0]
```

Out[473]=

```
False
```

## IGConnectedComponentSizes and IGWeaklyConnectedComponentSizes

In[474]:=

```
? IGConnectedComponentSizes
```

IGConnectedComponentSizes[graph] gives the sizes of graph's connected components in decreasing order.

In[475]:=

```
? IGWeaklyConnectedComponentSizes
```

IGWeaklyConnectedComponentSizes[graph] gives the sizes of graph's weakly connected components in decreasing order.

`IGWeaklyConnectedComponentsSizes` and `IGConnectedComponentSizes` return the sizes of the graph’s weakly or strongly connected components in decreasing order.

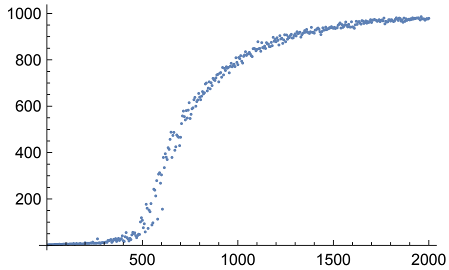
In large graphs, these functions will be faster than the equivalent `Length /@ ConnectedComponents[g]`.

The emergence of a giant component as the number of edges in a random graph increases.

In[476]:=

```
Table[
  {m, First@IGConnectedComponentSizes@RandomGraph[{1000, m}]},
  {m, 5, 2000, 5}
 ] // ListPlot
```

Out[476]=

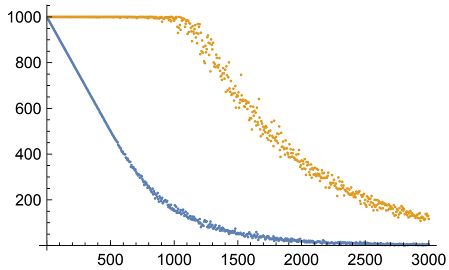



The number of weakly and strongly connected components versus the number of edges in a random directed graph.

In[477]:=

```
Table[
   With[{g = RandomGraph[{1000, m}, DirectedEdges → True]},
    {{m, Length@IGWeaklyConnectedComponentSizes[g]}, {m, Length@IGConnectedComponentSizes[g]}}
   ],
   {m, 5, 3000, 5}
  ] // Transpose // ListPlot
```

Out[477]=




## IGFindMinimumCuts

In[478]:=

```
? IGFindMinimumCuts
```

IGFindMinimumCuts[graph, s, t] gives all minimum edge cuts that disconnect s and t in a weighted graph.

`IGFindMinimalCuts[g, s, t]` finds all smallest-weight (i.e. minimum) edge cuts that disconnect vertex `t` from vertex `s`.

In[479]:=

```
g = ExampleData[{"NetworkGraph", "MetabolicNetworkAeropyrumPernix"}];
IGFindMinimumCuts[g, 30, 160]
```

Out[480]=

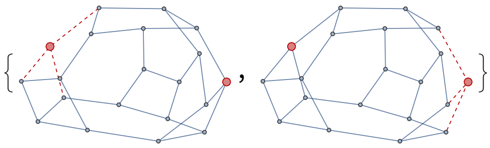
```
{{30 ⟷ 1 000 177, 30 ⟷ 1 000 178}, {30 ⟷ 1 000 178, 1 000 177 ⟷ 10}, {1 000 080 ⟷ 160, 1 000 092 ⟷ 160}}
```

Visualize all minimum cuts between two vertices in a random cubic graph.

In[481]:=

```
g = IGKRegularGame[20, 3];
HighlightGraph[g, Join[#, {1, 20}], GraphHighlightStyle → "Dotted", VertexSize → Large] & /@
 IGFindMinimumCuts[g, 1, 20]
```

Out[482]=



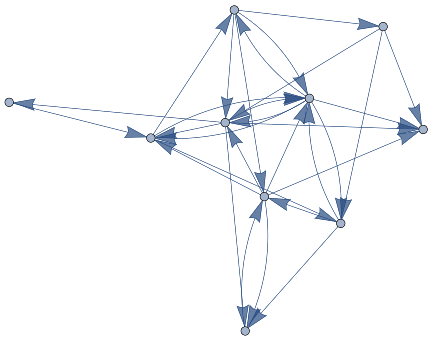
**Warning:** `IGFindMinimumCuts` takes edge weights into account, but it is only safe to use with integer weights. If the weights are not integers, then numerical roundoff errors may prevent the function from detecting that two cuts have the same total weight.

Create an integer-weighted graph with more than one minimum cut between vertices `1` and `10`:

In[483]:=

```
g = IGTryUntil[Length@IGFindMinimumCuts[#, 1, 10] > 2 &][
  RandomGraph[{10, 30}, DirectedEdges → True, EdgeWeight → RandomInteger[{1, 10}, 30]]]
```

Out[483]=



In[484]:=

```
IGFindMinimumCuts[g, 1, 10]
```

Out[484]=

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

Multiplying the weights by `0.1` causes `IGFindMinimumCuts` to return fewer results because some of the weights are no longer exactly representable in binary:

In[485]:=

```
IGFindMinimumCuts[IGEdgeMap[0.1 # &, EdgeWeight, g], 1, 10]
```

Out[485]=

```
{{1 → 3, 1 → 6, 1 → 7, 1 → 8}}
```

If only a single minimum cut is needed, use `IGMinimumCut`:

In[486]:=

```
IGMinimumCut[g, 1, 10]
```

Out[486]=

```
{2 → 10, 3 → 10, 4 → 10}
```

The size (total weight) of the cut can be obtained with `IGMinimumCutValue`:

In[487]:=

```
IGMinimumCutValue[g, 1, 10]
```

Out[487]=

```
14.
```

### References


- J. S. Provan and D. R. Shier: A Paradigm for listing (s,t)-cuts in graphs, Algorithmica 15, 351--372, 1996.


## IGFindMinimalCuts

In[488]:=

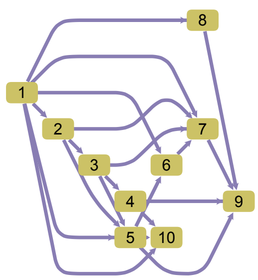
```
? IGFindMinimalCuts
```

IGFindMinimalCuts[graph, s, t] gives all minimal edge cuts that disconnect s and t in graph.

`IGFindMinimalCuts[g, s, t]` finds all *unweighted* minimal edge cuts that disconnect vertex `t` from vertex `s`.

In[489]:=

```
g =
```




```
;
```

In[490]:=

```
IGFindMinimalCuts[g, 1, 10]
```

Out[490]=

```
{{1 → 2, 1 → 5, 1 → 10}, {1 → 2, 1 → 10, 5 → 10}, {1 → 5, 1 → 10, 2 → 3, 2 → 5},
 {1 → 10, 2 → 3, 5 → 10}, {1 → 5, 1 → 10, 2 → 5, 3 → 4, 3 → 5}, {1 → 10, 3 → 4, 5 → 10},
 {1 → 5, 1 → 10, 2 → 5, 3 → 5, 4 → 10}, {1 → 10, 4 → 10, 5 → 10}}
```

The set of all minimum cuts is a subset of the minimal ones.

In[491]:=

```
IGFindMinimumCuts[g, 1, 10]
```

Out[491]=

```
{{1 → 2, 1 → 5, 1 → 10}, {1 → 2, 1 → 10, 5 → 10},
 {1 → 10, 2 → 3, 5 → 10}, {1 → 10, 3 → 4, 5 → 10}, {1 → 10, 4 → 10, 5 → 10}}
```

In[492]:=

```
SubsetQ[%%, %]
```

Out[492]=

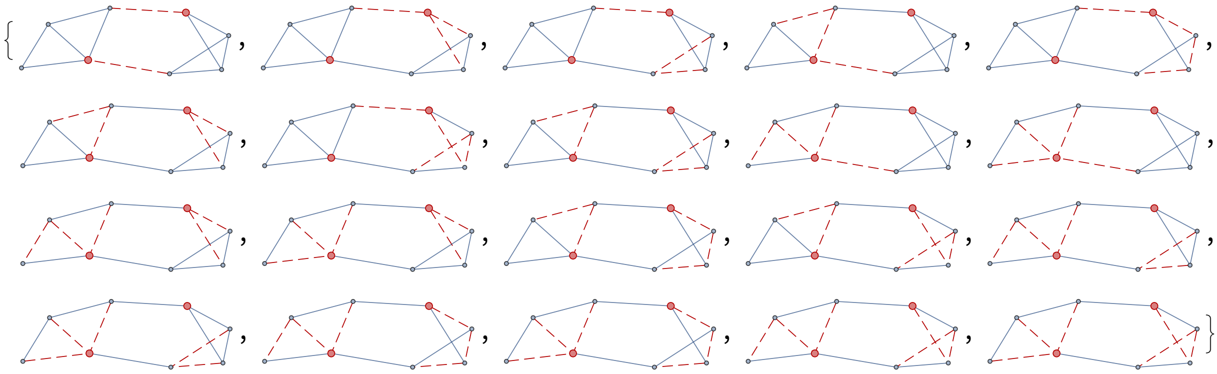
```
True
```

Visualize all minimal cuts between two vertices, from smallest to largest, in an undirected graph.

In[493]:=

```
g = IGGiantComponent@RandomGraph[{8, 12}];
HighlightGraph[g, Join[#, {1, 8}], GraphHighlightStyle → "Dashed", VertexSize → Medium] & /@
 SortBy[Length]@IGFindMinimalCuts[g, 1, 8]
```

Out[494]=



### References

- J. S. Provan and D. R. Shier: A Paradigm for listing (s,t)-cuts in graphs, Algorithmica 15, 351--372, 1996.

## Vertex separators

A vertex separator is a set of vertices whose removal disconnects the graph.

In[495]:=

```
? IGMinimalSeparators
```

IGMinimalSeparators[graph] gives all minimal separator vertex sets. A vertex set is a separator if its removal disconnects the graph. Edge directions are ignored.

In[496]:=

```
? IGMinimumSeparators
```

IGMinimumSeparators[graph] gives all separator vertex sets of minimum size. A vertex set is a separator if its removal disconnects the graph. Edge directions are ignored.

In[497]:=

```
? IGVertexSeparatorQ
```

IGVertexSeparatorQ[graph, {vertex1, vertex2, …}] tests if the given set of vertices disconnects the graph. Edge directions are ignored.

In[498]:=

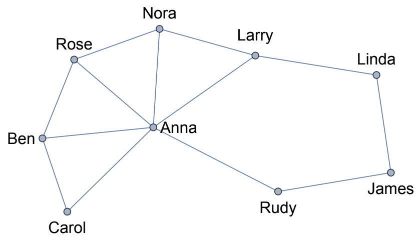
```
? IGMinimalVertexSeparatorQ
```

IGMinimalVertexSeparatorQ[graph, {vertex1, vertex2, …}] tests if the given vertex set is a minimal separator. Edge directions are ignored.

In[499]:=

```
g = ExampleData[{"NetworkGraph", "Friendship"}]
```

Out[499]=




In[500]:=

```
separators = IGMinimumSeparators[g]
```

Out[500]=

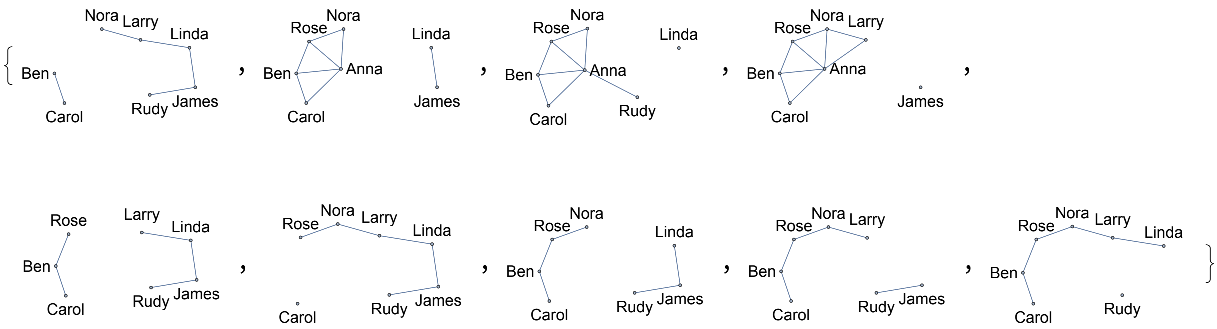
```
{{Anna, Rose}, {Larry, Rudy}, {Larry, James}, {Rudy, Linda},
 {Anna, Nora}, {Anna, Ben}, {Anna, Larry}, {Anna, Linda}, {Anna, James}}
```

Removing any of these vertex sets will disconnect the graph:

In[501]:=

```
VertexDelete[g, #] & /@ separators
```

Out[501]=



In[502]:=

```
IGVertexSeparatorQ[g, #] & /@ separators
```

Out[502]=

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

In[503]:=

```
IGMinimalVertexSeparatorQ[g, #] & /@ separators
```

Out[503]=

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

Removing Anna, Nora and Larry also disconnects the graph, thus this vertex set is a separator:

In[504]:=

```
IGVertexSeparatorQ[g, {"Anna", "Nora", "Larry"}]
```

Out[504]=

```
True
```

But it is not minimal:

In[505]:=

```
IGMinimalVertexSeparatorQ[g, {"Anna", "Nora", "Larry"}]
```

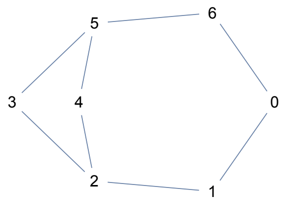
Out[505]=

```
False
```

`IGMinimumSeparators` returns only those vertex separators which are of the smallest possible size in the graph. `IGMinimalSeparators` returns all separators which cannot be made smaller by removing a vertex from them. The former is a subset of the latter.

In[506]:=

```
g =
```




```
;
```

In[507]:=

```
IGMinimalSeparators[g]
```

Out[507]=

```
{{1, 6}, {0, 2}, {1, 3, 4}, {2, 5}, {3, 4, 6}, {0, 5}, {2, 6}, {1, 5}, {0, 3, 4}}
```

In[508]:=

```
IGMinimumSeparators[g]
```

Out[508]=

```
{{2, 5}, {1, 5}, {1, 6}, {0, 5}, {2, 6}, {0, 2}}
```

In[509]:=

```
SubsetQ[%%, %]
```

Out[509]=

```
True
```

## IGEdgeConnectivity

In[510]:=

```
? IGEdgeConnectivity
```

IGEdgeConnectivity[graph] gives the smallest number of edges whose deletion disconnects graph.
IGEdgeConnectivity[graph, s, t] gives the smallest number of edges whose deletion disconnects vertices s and t in graph.

`IGEdgeConnectivity` ignores edge weights. To take edge weights into account, use `IGMinimumCutValue` instead.

Compute the edge connectivity of the dodecahedral graph.

In[511]:=

```
IGEdgeConnectivity[GraphData["DodecahedralGraph"]]
```

Out[511]=

```
3
```

The edge connectivity of the singleton graph is returned as 0.

In[512]:=

```
IGEdgeConnectivity[IGEmptyGraph[1]]
```

Out[512]=

```
0
```

## IGVertexConnectivity

In[513]:=

```
? IGVertexConnectivity
```

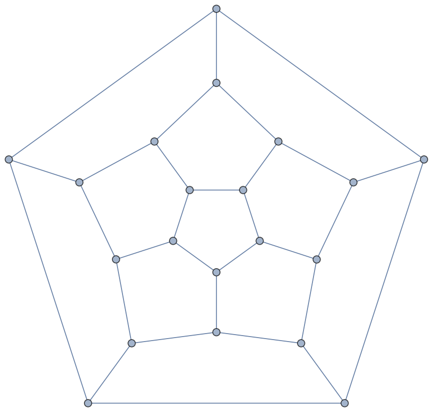
IGVertexConnectivity[graph] gives the smallest number of vertices whose deletion disconnects graph.
IGVertexConnectivity[graph, s, t] gives the smallest
number of vertices whose deletion disconnects vertices s and t in graph.

According to Steinitz's theorem, the skeleton of every convex polyhedron is a 3-vertex-connected planar graph.

In[514]:=

```
g = GraphData["DodecahedralGraph"]
```

Out[514]=



In[515]:=

```
IGVertexConnectivity[g]
```

Out[515]=

```
3
```

To find the specific vertex sets that disconnect the graph, use `IGMinimumSeparators` or `IGMinimalSeparators`.

In[516]:=

```
IGMinimumSeparators[g]
```

Out[516]=

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

The vertex connectivity of the singleton graph is returned as 0.

In[517]:=

```
IGVertexConnectivity[IGEmptyGraph[1]]
```

Out[517]=

```
0
```

## IGBiconnectedQ

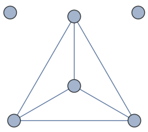
In[518]:=

```
? IGBiconnectedQ
```

IGBiconnectedQ[graph] tests if graph is biconnected.

`IGBiconnectedQ` checks if a graph is biconnected. Edge directions are ignored.

In[519]:=

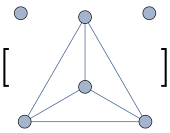
IGBiconnectedQ[  ]

Out[519]=

```
False
```

Since `IGBiconnectedComponents` does not return any isolated vertices, `Length@IGBiconnectedComponents[g] == 1` cannot be used to check if a graph is biconnected. Use `IGBiconnectedQ` instead.

In[520]:=

```
IGBiconnectedComponents[
```

```
]
```

Out[520]=

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

The singleton graph is not considered to be biconnected, but the two-vertex complete graph is.

In[521]:=

```
Table[IGBiconnectedQ@CompleteGraph[k], {k, 1, 2}]
```

Out[521]=

```
{False, True}
```

## IGBiconnectedComponents and IGBiconnectedEdgeComponents

In[522]:=

```
? IGBiconnectedComponents
```

IGBiconnectedComponents[graph] gives the vertices of the maximal biconnected subgraphs of graph. A graph is biconnected if the removal of any single vertex does not disconnect it. Isolated vertices are not returned.

In[523]:=

```
? IGBiconnectedEdgeComponents
```

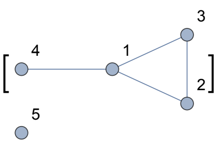
IGBiconnectedEdgeComponents[graph] gives the edges of the maximal biconnected subgraphs of graph. A graph is biconnected if the removal of any single vertex does not disconnect it.

`IGBiconnectedComponents` returns the vertices of the maximal biconnected components of the graph. `IGBiconnectedEdgeComponents` returns the edges of the components. Edge directions are ignored and isolated vertices are excluded.

`IGBiconnectedComponents` is equivalent to `KVertexConnectedComponents[…, 2]`, except that isolated vertices are not returned as individual components.

The articulation vertices will be part of more than a single component, thus the biconnected components are not disjoint subsets of the vertex set.

In[524]:=

```
IGBiconnectedComponents[
```


```
]
```

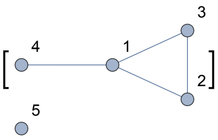
Out[524]=

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

However, each edge is part of precisely one biconnected components.

In[525]:=

IGBiconnectedEdgeComponents[




]

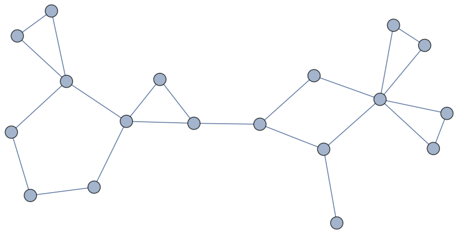
Out[525]=

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

Thus, visualizing biconnected components is best done by colouring the edges, not the vertices.

In[526]:=

g =

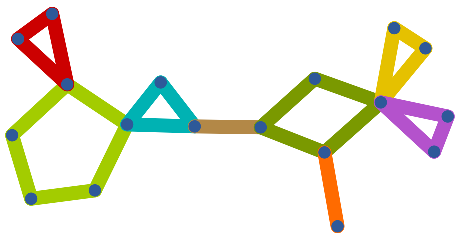


;

In[527]:=

```
HighlightGraph[g, IGBiconnectedEdgeComponents[g], GraphStyle → "ThickEdge"]
```

Out[527]=



## IGArticulationPoints

In[528]:=

```
? IGArticulationPoints
```

IGArticulationPoints[graph] gives the articulation points of graph. A vertex is an articulation point if its removal increases the number of (weakly) connected components in the graph.

`IGArticulationPoints` finds vertices whose removal increases the number of (weakly) connected components in the graph. Edge directions are ignored.

In[529]:=

```
g = Graph[{1 → 2, 2 → 3, 3 → 1, 3 → 4, 4 → 5, 5 → 6, 6 → 4},
  DirectedEdges → False, VertexLabels → Automatic]
```

Out[529]=


In[530]:=

```
IGArticulationPoints[g]
```

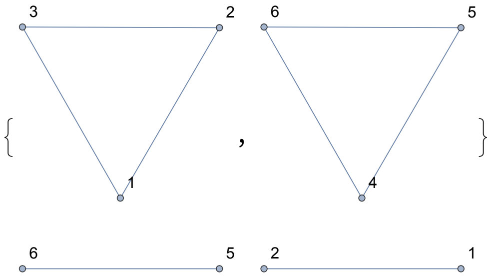
Out[530]=

```
{4, 3}
```

In[531]:=

```
VertexDelete[g, #] & /@ %
```

Out[531]=




Articulation points are also size-1 separators.

In[532]:=

```
IGMinimumSeparators[g]
```

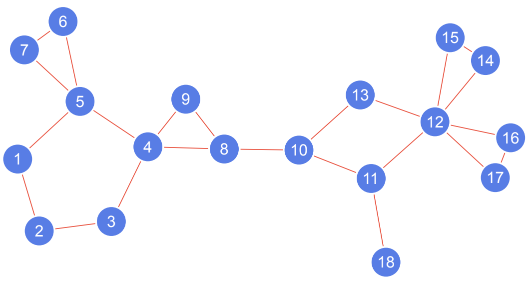
Out[532]=

```
{{4}, {3}}
```

Highlight the articulation points of a cactus graph.

In[533]:=

g =

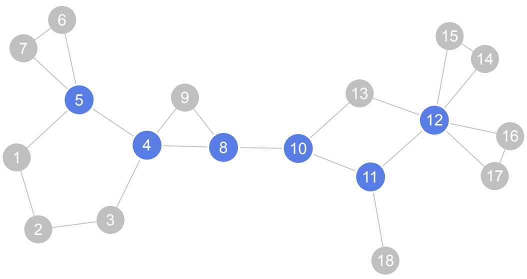



;

In[534]:=

```
HighlightGraph[g, IGArticulationPoints[g]]
```

Out[534]=

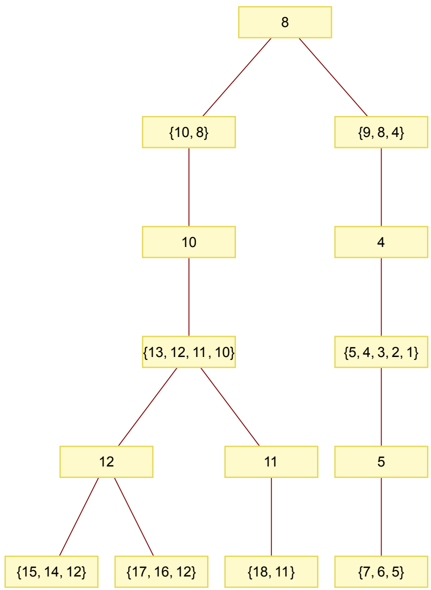


Compute the block-cut tree of a connected graph. The *blocks* are the biconnected components. Together with the articulation vertices they form a bipartite graph, specifically a tree.

In[535]:=

```
RelationGraph[
 MemberQ,
 Join[IGBiconnectedComponents[g], IGArticulationPoints[g]],
 DirectedEdges → False,
 GraphStyle → "ClassicDiagram",
 VertexSize → {3, 1} / 7, VertexLabelStyle → 8
]
```

Out[535]=




## IGBridges

In[536]:=

```
? IGBridges
```

IGBridges[graph] gives the bridges of graph. A bridge is an edge
 whose removal increases the number of (weakly) connected components in the graph.

A bridge is an edge whose removal disconnects the graph (or increases the number of connected components if the graph was already disconnected). Edge directions are ignored.

In[537]:=

```
IGShorthand["1-2-3-1-4-5-6-4"]
```

Out[537]=


In[538]:=

```
IGBridges[%]
```

Out[538]=

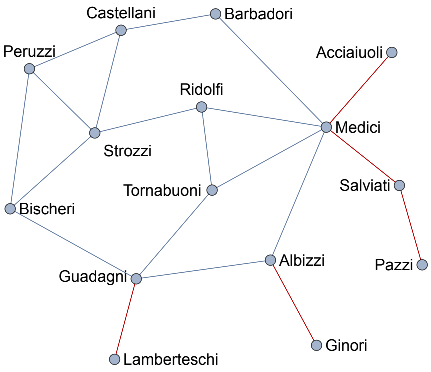
{1 ⟷ 4}

Highlight bridges in a network.

In[539]:=

```
g = ExampleData[{"NetworkGraph", "FlorentineFamilies"}];
HighlightGraph[g, IGBridges[g]]
```

Out[540]=




## IGSourceVertexList and IGSinkVertexList

In[541]:=

```
? IGSourceVertexList
```

IGSourceVertexList[graph] gives the list of vertices with no incoming connections.

In[542]:=

```
? IGSinkVertexList
```

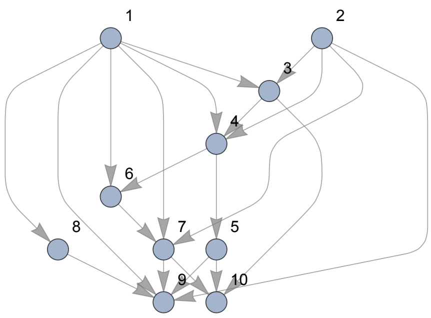
IGSinkVertexList[graph] gives the list of vertices with no outgoing connections.

Find and highlight the source and sink vertices of a random acyclic graph.

In[543]:=

```
g = DirectedGraph[RandomGraph[{10, 20}], "Acyclic",
  VertexLabels → "Name", VertexSize → Large, EdgeStyle → Gray]
```

Out[543]=




In[544]:=

```
IGSourceVertexList[g]
```

Out[544]=

```
{1, 2}
```

In[545]:=

```
IGSinkVertexList[g]
```

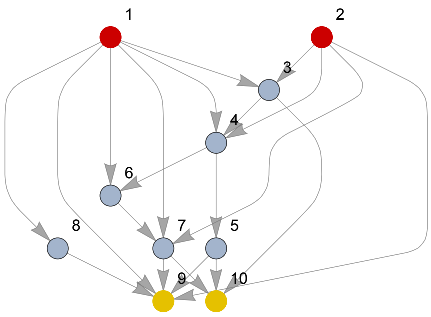
Out[545]=

```
{9, 10}
```

In[546]:=

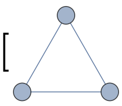
```
HighlightGraph[g, {IGSourceVertexList[g], IGSinkVertexList[g]}]
```

Out[546]=




Undirected graphs have neither source nor sink vertices because undirected edges are counted as bidirectional.

In[547]:=

```
IGSourceVertexList[
```

```
]
```

Out[547]=

```
{}
```

The exception is isolated vertices, which are counted both as sources and sinks.

In[548]:=

```
Through[{IGSourceVertexList, IGSinkVertexList}@Graph[{1, 2}, {}]]
```

Out[548]=

```
{{1, 2}, {1, 2}}
```

These are merely convenience functions that can be implemented straightforwardly as

In[549]:=

```
Pick[VertexList[g], VertexOutDegree[g], 0]
```

Out[549]=

```
{9, 10}
```

## IGIsolatedVertexList

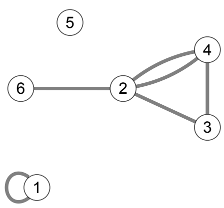
In[550]:=

```
? IGIsolatedVertexList
```

IGIsolatedVertexList[graph] gives the list of isolated vertices.

`IGIsolatedVertexList` returns the vertices which form their own weakly connected components. This includes vertices with no connections, as well as vertices with only self-loops.

In[551]:=

```
IGIsolatedVertexList[
```


```
]
```

Out[551]=

```
{1, 5}
```

## IGGiantComponent

In[552]:=

```
? IGGiantComponent
```

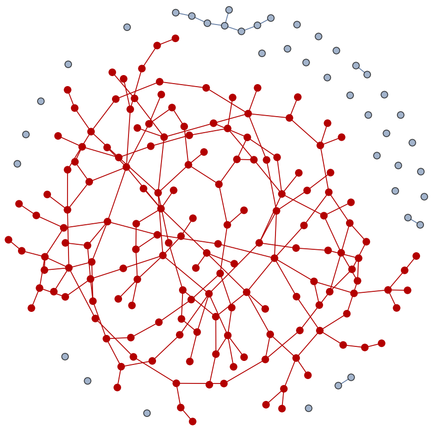
IGGiantComponent[graph] gives the largest weakly connected component of graph.

`IGGiantComponent` is a convenience function that returns the largest weakly connected component of graph. If there are multiple components of largest size, there is no guarantee about which one would be returned. If this is a concern, use `WeaklyConnectedComponents` or `WeaklyConnectedGraphComponents` instead.

In[553]:=

```
g = RandomGraph[{200, 200}];
HighlightGraph[
  g,
  IGGiantComponent[g]
 ] // IGLayoutFruchtermanReingold
```

Out[554]=

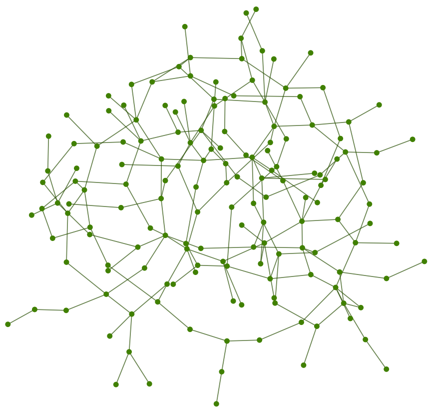


`IGGiantComponent` takes all standard graph options.

In[555]:=

```
IGGiantComponent[g, GraphStyle → "BasicGreen", GraphLayout → "SpringEmbedding"]
```

Out[555]=

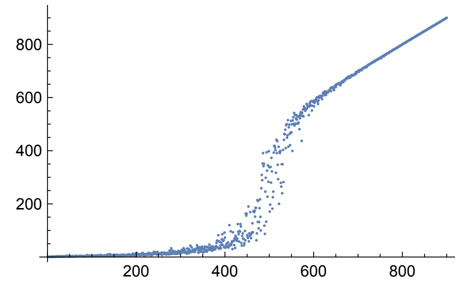


Size of the giant component of a random subgraph of a grid graph.

In[556]:=

```
g = IGSquareLattice[{30, 30}, "Periodic" → True];
Table[
  {k, VertexCount@IGGiantComponent@Subgraph[g, RandomSample[VertexList[g], k]]},
  {k, 1, VertexCount[g], 1}
 ] // ListPlot
```

Out[557]=




## IGPercolationCurve

In[558]:=

```
? IGPercolationCurve
```

IGPercolationCurve[graph] gives a percolation curve corresponding to random edge removal, as {meanDegree, largestComponentFraction} pairs.
IGPercolationCurve[edges] gives the percolation curve when edges are added in the specified order.
IGPercolationCurve[edges, n] assumes that there are n vertices.

**Experimental:** This is experimental functionality that may change in the future.

`IGPercolationCurve` computes the percolation curve for a sequence of edge additions (interpretable as edge removals in reverse order). The `i`th element of the result is the mean degree and the fraction of vertices within the largest component before adding the `i`th edge.

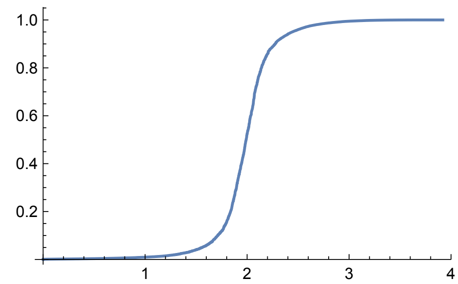
`IGPercolationCurve[graph]` is equivalent to `IGPercolationCurve[RandomSample@EdgeList[graph], VertexCount[graph]]`.

Plot the averaged percolation curve of a grid graph over many random edge removals.

In[559]:=

```
ListLinePlot@Mean@Table[IGPercolationCurve@GridGraph[{50, 50}], {100}]
```

Out[559]=




Percolation curve for a random geometric graph when edges are removed in order of decreasing or increasing betweenness.

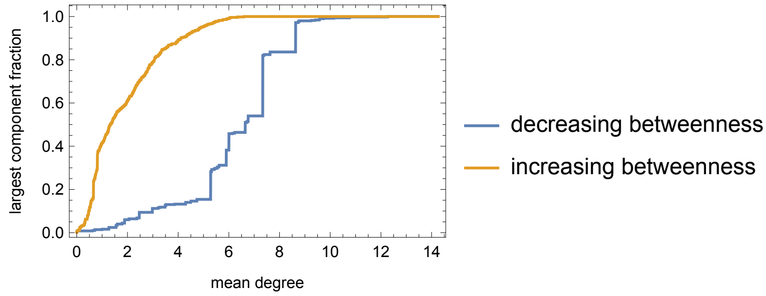
In[560]:=

```
g = IGGeometricGame[500, 0.1];
edgeOrder = EdgeList[g]〚Ordering@IGEdgeBetweenness[g]〛;
```

In[562]:=

```
ListLinePlot[IGPercolationCurve /@ {edgeOrder, Reverse[edgeOrder]},
 PlotLegends → {"decreasing betweenness", "increasing betweenness"},
 FrameLabel → {"mean degree", "largest component fraction"},
 Frame → True, PlotRange → All]
```

Out[562]=




IGPercolationCurve also accepts a list of pairs in addition to a list of edge expressions.

In[563]:=

```
IGPercolationCurve@RandomInteger[{1, 10}, {20, 2}]
```

Out[563]=

```
{{0., 0.1}, {0., 0.2}, {0.2, 0.2}, {0.4, 0.2}, {0.6, 0.2}, {0.8, 0.3},
 {1., 0.4}, {1.2, 0.4}, {1.4, 0.6}, {1.6, 0.7}, {1.8, 0.7}, {2., 0.9}, {2.2, 0.9},
 {2.4, 1.}, {2.6, 1.}, {2.8, 1.}, {3., 1.}, {3.2, 1.}, {3.4, 1.}, {3.6, 1.}, {3.8, 1.}}
```

IGPercolationCurve works efficiently on large networks.

In[564]:=

```
g = ExampleData[{"NetworkGraph", "WorldWideWeb"}];
EdgeCount[g]
```

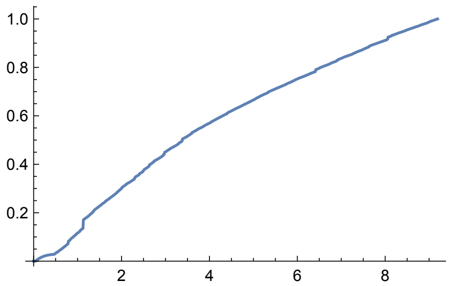
Out[565]=

1 497 134

In[566]:=

```
ListLinePlot[IGPercolationCurve[g], MaxPlotPoints → 1000]
```

Out[566]=




## Trees

A tree is a connected graph that contains no undirected cycles.

## IGTreeQ

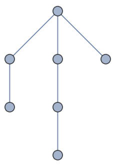
In[567]:=

```
? IGTreeQ
```

IGTreeQ[graph] tests if graph is a tree or out–tree.
IGTreeQ[graph, "Out"] tests if graph is an out–tree (arborescence).
IGTreeQ[graph, "In"] tests if graph is an in–tree (anti–arborescence).
IGTreeQ[graph, "All"] ignores edge directions during the test.

`IGTreeQ` checks if a graph is a tree. An undirected tree is a connected graph with no cycles. A directed tree is similar, with its edges oriented either away from a root vertex (out-tree or arborescence) or towards a root vertex (in-tree or anti-arborescence).

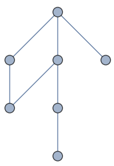
In[568]:=

`IGTreeQ[`  `]`

Out[568]=

```
True
```

In[569]:=

`IGTreeQ[`  `]`

Out[569]=

```
False
```

By convention, the null graph is not a tree.

In[570]:=

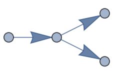
```
IGTreeQ[IGEmptyGraph[0]]
```

Out[570]=

```
False
```

This is an out-tree.

In[571]:=

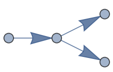
`IGTreeQ[`  `]`

Out[571]=

```
True
```

It is not also an in-tree.

In[572]:=

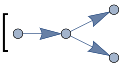
`IGTreeQ[`  `, "In"]`

Out[572]=

```
False
```

It becomes an in-tree if we reverse its edges.

In[573]:=

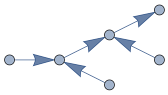
**IGTreeQ[ReverseGraph[****], "In"]**

Out[573]=

```
True
```

This graph is neither an out-tree nor an in-tree.

In[574]:=

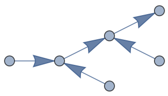
**IGTreeQ[****]**

Out[574]=

```
False
```

However, it becomes a tree if we ignore edge directions.

In[575]:=

**IGTreeQ[****, "All"]**

Out[575]=

```
True
```

## IGForestQ

In[576]:=

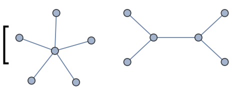
```
? IGForestQ
```

IGForestQ[graph] tests if graph is a forest of trees or out–trees.
IGForestQ[graph, "Out"] tests if graph is a forest of out–trees (arborescences).
IGForestQ[graph, "In"] tests if graph is a forest of in–trees (anti–arborescences).
IGForestQ[graph, "All"] ignores edge directions during the test.

`IGForestQ` is a convenience function that tests if all connected components of a graph are trees.

This graph is not a tree, but it is a forest.

In[577]:=

**Through[{IGTreeQ, IGForestQ}[****]]**

Out[577]=

```
{False, True}
```

By convention, the null graph is not a tree, but it is a forest.

In[578]:=

```
{IGTreeQ[IGEmptyGraph[0]], IGForestQ[IGEmptyGraph[0]]}
```

Out[578]=

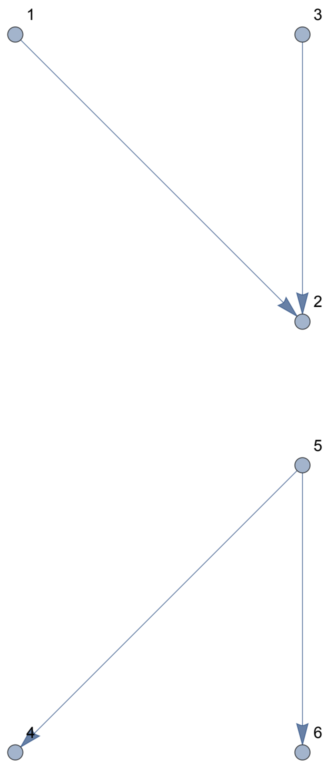
```
{False, True}
```

Use the second argument to test for forests of out-trees or in-trees. By default, directed graphs are checked to be out-forests.

In[579]:=

```
g = IGShorthand["1->2<-3, 4<-5->6"]
```

Out[579]=




In[580]:=

```
{IGForestQ[g], IGForestQ[g, "Out"], IGForestQ[g, "In"], IGForestQ[g, "All"]}
```

Out[580]=

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

## IGStrahlerNumber

In[581]:=

```
? IGStrahlerNumber
```

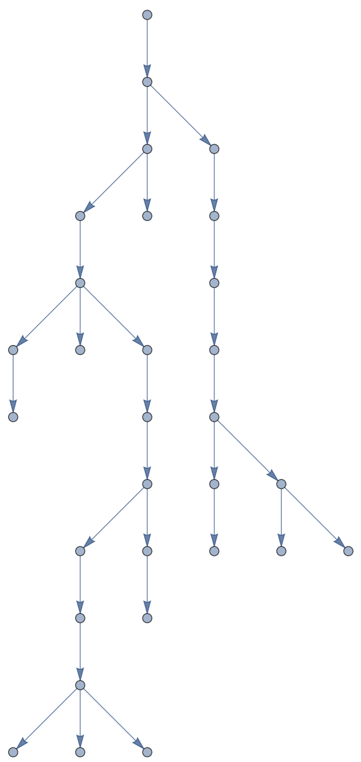
IGStrahlerNumber[tree] gives the Horton–Strahler number of each vertex in a directed out–tree.

`IGStrahlerNumber` computes the Horton–Strahler index of each vertex in a rooted tree. The tree must be directed—this is how the root is encoded. The Horton–Strahler index of the tree itself is the index of the root, i.e. the largest returned index. This measure is also called *stream order*, as it was originally used to characterize river networks.

In[582]:=

```
tree = IGTreeGame[30, DirectedEdges → True, GraphLayout → "LayeredDigraphEmbedding"]
```

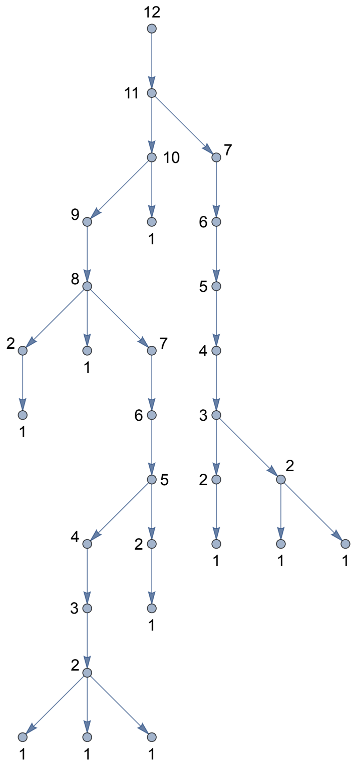
Out[582]=



In[583]:=

```
IGVertexMap[# &, VertexLabels → IGStrahlerNumber, tree]
```

Out[583]=



To get the Horton–Strahler number of the tree, find the maximal element.

In[584]:=

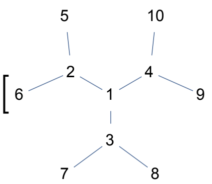
```
Max@IGStrahlerNumber[tree]
```

Out[584]=

```
12
```

`IGStrahlerNumber` requires a directed (i.e. rooted) tree as input.

In[585]:=

**IGStrahlerNumber[**  **]**



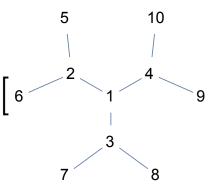
IGraphM: strahlerNumber: the graph is not a directed out-tree.

Out[585]=

```
$Failed
```

Orient undirected trees, effectively specifying a root vertex, before passing them to `IGStrahlerNumber`.

In[586]:=

**IGStrahlerNumber@IGOrientTree[**  **, 5]**



Out[586]=

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

## IGTreelikeComponents

In[587]:=

```
? IGTreelikeComponents
```

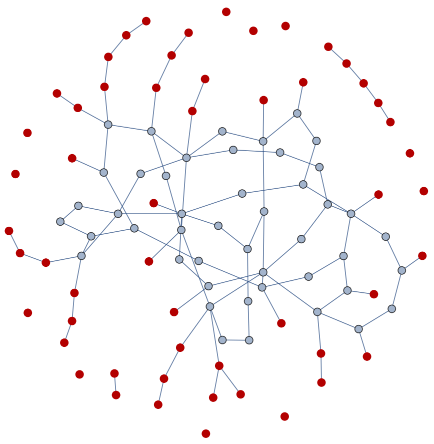
IGTreelikeComponents[graph] returns the vertices that make up tree-like components.

IGTreelikeComponents finds the tree-like components of an undirected graph by repeatedly identifying and removing degree-1 vertices. Vertices in the tree-like components are not part of any undirected cycle, nor are they on a path connecting vertices that belong to a cycle.

In[588]:=

```
g = RandomGraph[{100, 100}];
HighlightGraph[
  g,
  IGTreelikeComponents[g]
 ] // IGLayoutFruchtermanReingold
```

Out[589]=

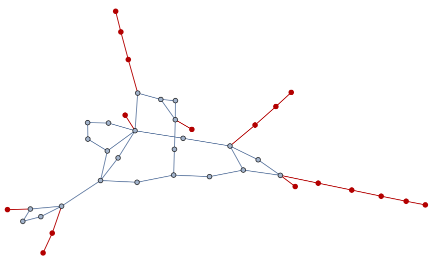


Highlight both the edges and vertices of tree-like components.

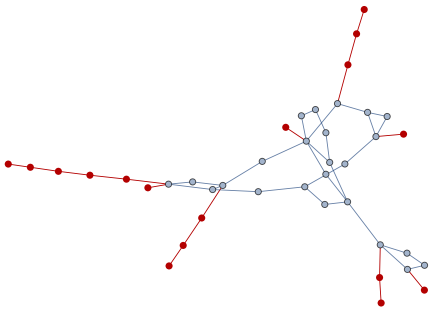
In[590]:=

```
g = IGGiantComponent@RandomGraph[{50, 50}];
HighlightGraph[
 g,
 Join[
  Union @@ (IncidenceList[g, #] &) /@ IGTreelikeComponents[g],
  IGTreelikeComponents[g]
 ]
]
```

Out[591]=

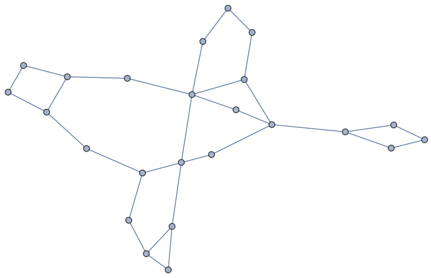


In[592]:=
```
IGLayoutFruchtermanReingold[%]
```
Out[592]=

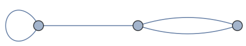


Remove tree-like components.

In[593]:=
```
VertexDelete[g, IGTreelikeComponents[g]]
```
Out[593]=



Vertices incident to multi-edges or loop-edges are not part of tree-like components.

In[594]:=
```
IGTreelikeComponents[
```

```
]
```
Out[594]=
```
{1}
```

## IGFromPrufer

In[595]:=
```
? IGFromPrufer
```

IGFromPrufer[sequence] constructs a tree from a Prüfer sequence.

## IGToPrufer

In[596]:=
```
? IGToPrufer
```

IGToPrufer[tree] gives the Prüfer sequence of a tree.

# Spanning trees

A spanning tree of a graph is a subgraph that is a tree and contains all the graph’s vertices.

## IGSpanningTree

In[597]:=

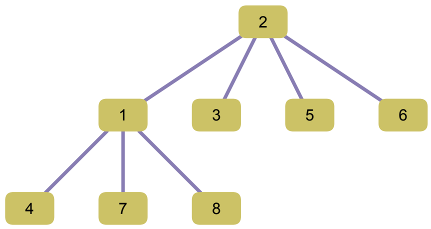
```
? IGSpanningTree
```

IGSpanningTree[graph] gives a minimum spanning tree of graph. Edge directions are ignored. Edge weights are taken into account and are preserved in the tree.

In[598]:=

```
IGSpanningTree[RandomGraph[{8, 20}], GraphStyle → "DiagramGold"]
```

Out[598]=




Find the shortest set of paths connecting a set of points in the plane:

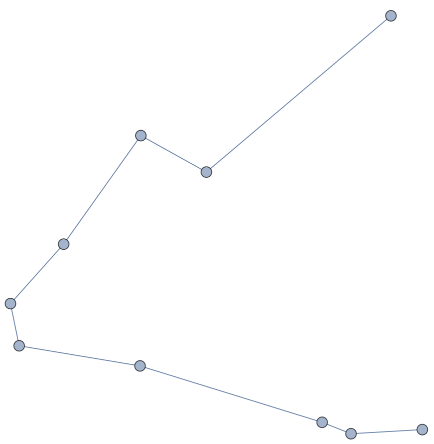
In[599]:=

```
pts = RandomReal[1, {10, 2}];
g = IGMeshGraph@DelaunayMesh[pts];
```

In[601]:=

```
tree = IGSpanningTree[g, VertexCoordinates → pts]
```

Out[601]=



The edge weights are preserved in the result.

In[602]:=

```
IGEdgeWeightedQ[tree]
```

Out[602]=

```
True
```

Compute the total path length.

In[603]:=

```
Total@IGEdgeProp[EdgeWeight][tree]
```

Out[603]=

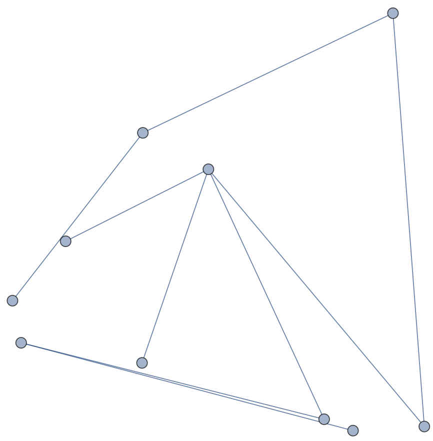
```
2.0021
```

Find a *maximum* spanning tree by negating the weights before running the algorithm.

In[604]:=

```
IGSpanningTree[IGEdgeMap[Minus, EdgeWeight, g], VertexCoordinates → pts]
```

Out[604]=

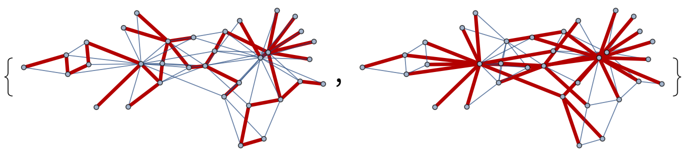


Find the minimum and maximum spanning trees of a network, using its edge betweenness as edge weights.

In[605]:=

```
g = ExampleData[{"NetworkGraph", "ZacharyKarateClub"}];
HighlightGraph[
   g,
   EdgeList@IGSpanningTree@IGEdgeMap[#, EdgeWeight → IGEdgeBetweenness, g],
   GraphHighlightStyle → "Thick", ImageSize → Small
  ] & /@ {
  Identity, (* minimum spanning tree *)
  Minus (* maximum spanning tree *)
 }
```

Out[606]=



## IGRandomSpanningTree

In[607]:=

```
? IGRandomSpanningTree
```

IGRandomSpanningTree[graph] gives a random spanning tree of graph. All spanning trees are generated with equal probability.
IGRandomSpanningTree[{graph, vertex}] gives a random spanning tree of the graph component containing vertex.
IGRandomSpanningTree[spec, n] gives a list of n random spanning trees.

`IGRandomSpanningTree` samples the spanning trees (or forests) of a graph uniformly by performing a loop-erased random walk. Edge directions are ignored.

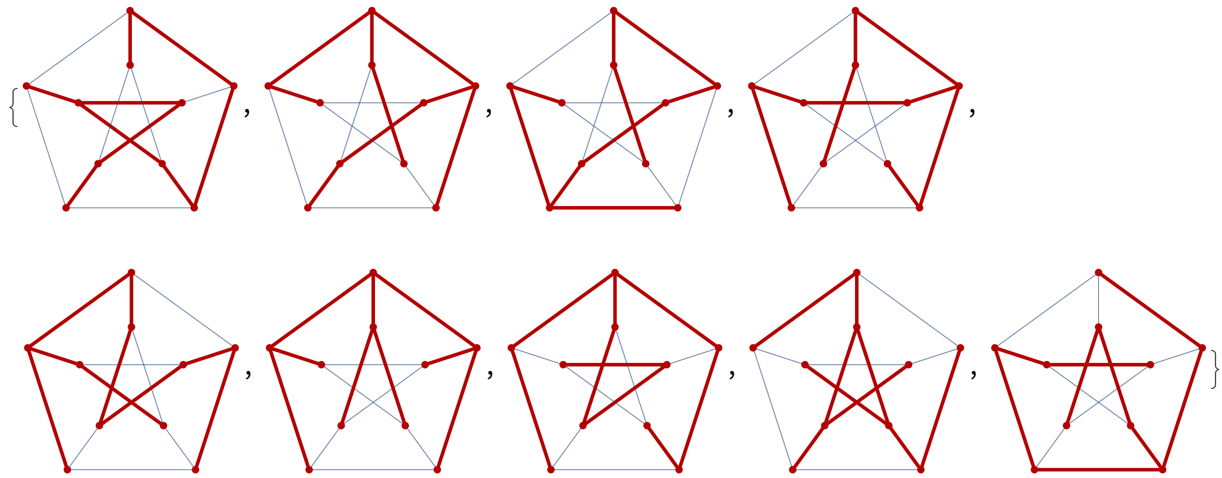
If a spanning forest of the entire graph is requested using `IGRandomSpanningTree[g]`, then the vertex names and ordering are preserved. If a spanning tree of only a single component is requested using `IGRandomSpanningTree[{g, v}]`, then this is not the case.

Highlight a few random spanning trees of the Petersen graph.

In[608]:=

```
g = PetersenGraph[];
HighlightGraph[g, #, GraphHighlightStyle → "Thick"] & /@ IGRandomSpanningTree[g, 9]
```

Out[609]=

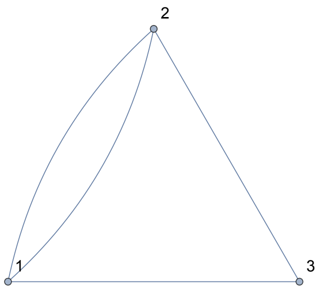


If the input is a multi-graph, each edge will be considered separately for the purpose of spanning tree calculations. Thus the following graph has not 3, but 5 different spanning trees. Two pairs of these are indistinguishable based on their adjacency matrix due to the indistinguishability of the two parallel 1 ⟷ 2 edges. However, since all 5 spanning trees are generated with equal probability, two of the 3 adjacency matrices will appear twice as frequently as the third one.

In[610]:=

```
g = IGShorthand["1-2-3-1,1-2", MultiEdges → True, ImageSize → Small]
```

Out[610]=




In[611]:=

```
IGRandomSpanningTree[g, 10 000] // CountsBy[AdjacencyMatrix] // KeySort // KeyMap[MatrixForm]
```

Out[611]=

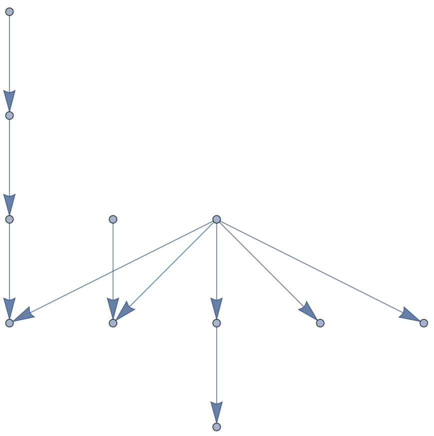
$$\left\langle\left| \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \to 1976, \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \to 3992, \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \to 4032 \right|\right\rangle$$

Edge directions are ignored for the purpose of spanning tree calculation. Thus the result may not be an out-tree.

In[612]:=

```
IGRandomSpanningTree@WheelGraph[11, DirectedEdges → True]
```

Out[612]=

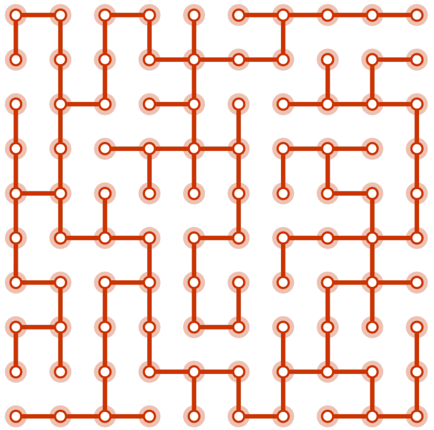


Create mazes by taking random spanning trees of grid graphs.

In[613]:=

```
g = GridGraph[{10, 10}, GraphStyle → "Web"];
HighlightGraph[g, IGRandomSpanningTree[g], GraphHighlightStyle → "DehighlightHide"]
```

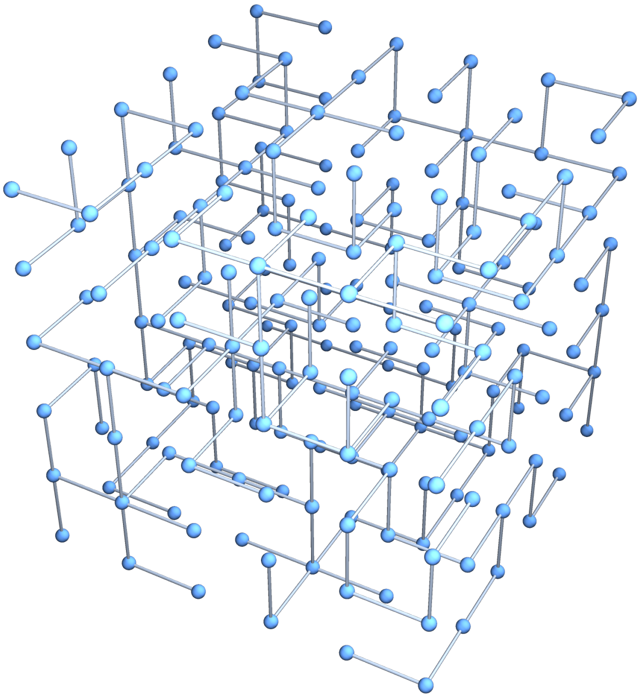
Out[614]=



In[615]:=

```
g = GridGraph[{6, 6, 6}, VertexCoordinates → Tuples[Range[6], {3}]];
HighlightGraph[g, IGRandomSpanningTree[g], GraphHighlightStyle → "DehighlightHide"]
```

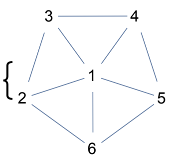
Out[616]=



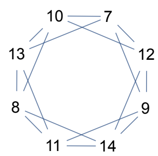
Generate a random spanning tree of the component containing vertex 8.

In[617]:=

IGRandomSpanningTree[{




,




, 8}, VertexLabels → Automatic]

Out[617]=


## IGSpanningTreeCount

In[618]:=

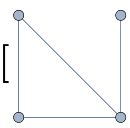
```
? IGSpanningTreeCount
```

IGSpanningTreeCount[graph] gives the number of spanning trees of graph.
IGSpanningTreeCount[graph, vertex] gives the number of spanning trees rooted in vertex for a directed graph.

`IGSpanningTreeCount` computes the number of spanning trees of a graph using Kirchhoff's theorem. Multigraphs and directed graphs are supported.

In[619]:=

IGSpanningTreeCount[



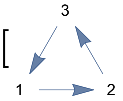
]

Out[619]=

3

The number of spanning trees of a directed graph, rooted in any vertex.

In[620]:=

`IGSpanningTreeCount[`  `]`

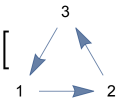


Out[620]=

`3`

The number of spanning trees rooted in vertex 1.

In[621]:=

`IGSpanningTreeCount[`  `, 1]`



Out[621]=

`1`

In[622]:=

```
IGSpanningTreeCount[PetersenGraph[]]
```

Out[622]=

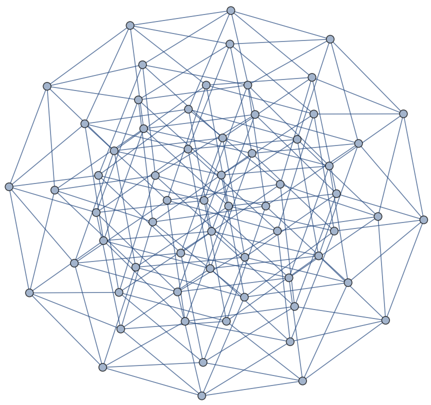
`2000`

`IGSpanningTreeCount` works on large graphs.

In[623]:=

```
g = HypercubeGraph[6]
```

Out[623]=



In[624]:=

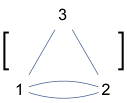
```
IGSpanningTreeCount[g]
```

Out[624]=

`1 657 509 127 047 778 993 870 601 546 036 901 052 416 000 000`

Edge multiplicities are taken into account. Thus the following graph has not 3, but 5 different spanning trees.

In[625]:=

`IGSpanningTreeCount[`  `]`



Out[625]=

`5`

## Dominance

In a directed graph, a vertex *d* is said to *dominate* a vertex *v* if every path from the root to *v* passes through *d*. We say that *d* is an *immediate dominator* of *v* if it does not dominate any other dominator of *v*.

A dominator tree of a graph consists of the same vertices as the graph, and the children of a vertex are those other vertices which it immediately dominates.

## IGDominatorTree

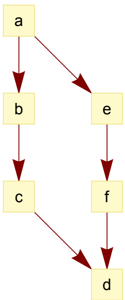
In[626]:=

```
? IGDominatorTree
```

IGDominatorTree[graph, root] returns the dominator tree of a directed graph starting from root.

Find the dominator tree of a directed graph.

In[627]:=

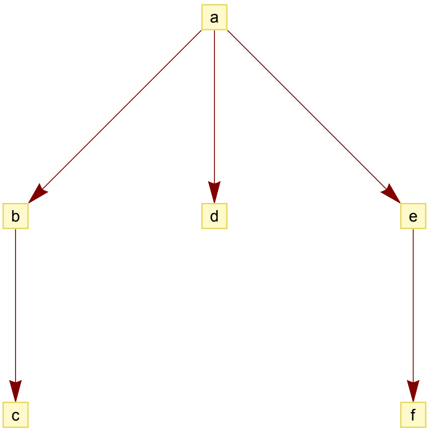
g =  ;



In[628]:=

```
IGDominatorTree[g, "a", GraphStyle → "VintageDiagram"]
```

Out[628]=

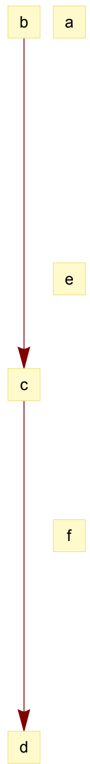


Vertices that cannot be reached from the specified root are left isolated in the returned graph.

In[629]:=

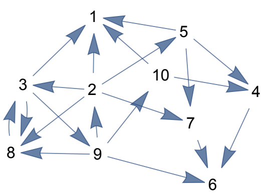
```
IGDominatorTree[g, "b", GraphStyle → "VintageDiagram"]
```

Out[629]=



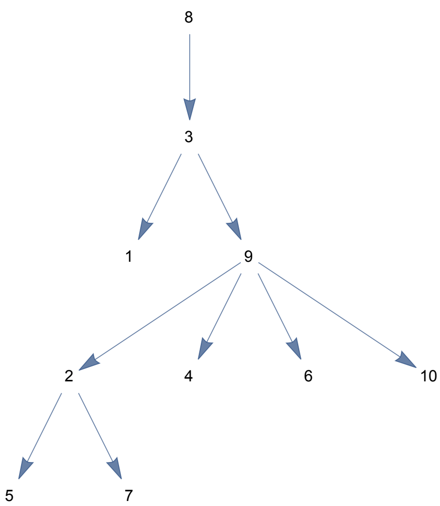

`IGDominatorTree` accepts all standard `Graph` options.

In[630]:=

```
IGDominatorTree[
```



```
, 8,
 VertexShapeFunction → "Name", PerformanceGoal → "Quality"]
```

Out[630]=



## IGImmediateDominators

In[631]:=

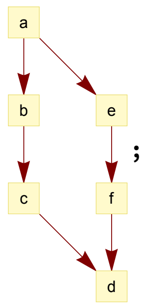
```
? IGImmediateDominators
```

IGImmediateDominators[graph, root] returns the immediate dominator of each vertex relative to root.

Directly find the immediate dominators of vertices in a graph.

In[632]:=

```
g = 
```




```
;
```

```
IGImmediateDominators[g, "a"]
```

Out[633]=

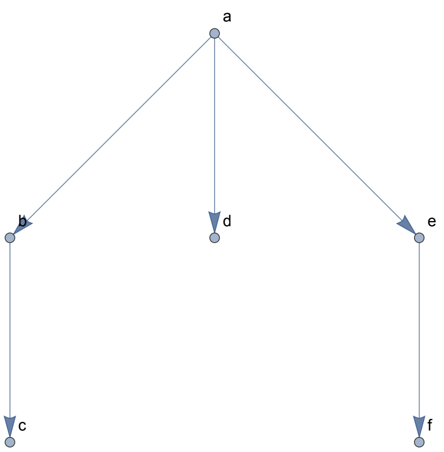
```
⟨|b → a, c → b, d → a, e → a, f → e|⟩
```

The immediate dominator of a vertex is its parent in the dominator tree.

In[634]:=

```
tree = IGDominatorTree[g, "a", VertexLabels → Automatic]
```

Out[634]=




In[635]:=

```
IGAdjacencyList[tree, "In"]
```

Out[635]=

```
⟨|a → {}, b → {a}, c → {b}, d → {a}, e → {a}, f → {e}|⟩
```

Neither the root, nor vertices unreachable from the root are included in the keys of the returned association.

In[636]:=

```
IGImmediateDominators[g, "b"]
```

Out[636]=

```
⟨|c → b, d → c|⟩
```

## *k*-cores

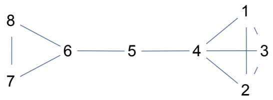
In[637]:=

```
? IGCoreness
```

IGCoreness[graph] returns the coreness of each vertex. Coreness is the highest order of a k–core containing the vertex.
IGCoreness[graph, "In"] considers only in–degrees in a directed graph.
IGCoreness[graph, "Out"] considers only out–degrees in a directed graph.

A *k*-core of a graph is a maximal subgraph in which each vertex has degree at least *k*. The coreness of a vertex is the highest order of *k*-cores that contain it.

In[638]:=

g =  ;



In[639]:=

```
IGCoreness[g]
```

Out[639]=

```
{3, 3, 3, 3, 2, 2, 2, 2}
```

In[640]:=

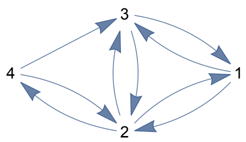
```
{KCoreComponents[g, 2], KCoreComponents[g, 3]}
```

Out[640]=

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

By default, edge directions are ignored, and multi-edges are considered.

In[641]:=

g =  ;



In[642]:=

```
IGCoreness[g]
```

Out[642]=

```
{4, 4, 4, 3}
```

Use the second argument to consider only in- or out-degrees.

In[643]:=

```
IGCoreness[g, "In"]
```

Out[643]=

```
{2, 2, 2, 1}
```

In[644]:=

```
IGCoreness[g, "Out"]
```

Out[644]=

```
{2, 2, 2, 2}
```

## Matchings

A matching of a graph is a subset of its edges that share no vertices between them.

### IGMaximumMatching

In[645]:=

```
? IGMaximumMatching
```

IGMaximumMatching[graph] gives a maximum matching of graph. Edge weights are ignored.

A matching of a graph is also known as an independent edge set.

`IGMaximumMatching` ignores edge directions and edge weights.

In[646]:=

```
g = RandomGraph[{10, 20}];
```

In[647]:=

```
IGMaximumMatching[g]
```

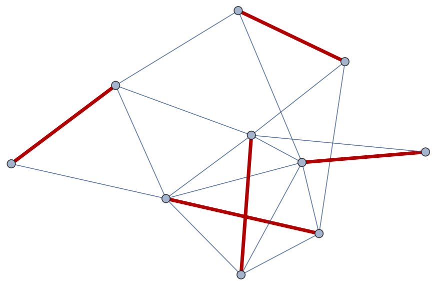
Out[647]=

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

In[648]:=

```
HighlightGraph[g, IGMaximumMatching[g], GraphHighlightStyle → "Thick"]
```

Out[648]=



### IGMatchingNumber

In[649]:=

```
? IGMatchingNumber
```

IGMatchingNumber[graph] gives the matching number of graph.

The matching number of a graph is the size of its maximum matchings.

## Graph traversal

### IGUnfoldTree

In[650]:=

```
? IGUnfoldTree
```

IGUnfoldTree[graph, {root1, root2, …}] performs a breadth–first search on graph starting from the given roots, and converts it to a tree or forest by replicating vertices that were found more than once. The original vertex that generated a tree node is stored in the "OriginalVertex" property.

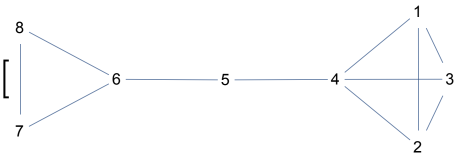
`IGUnfoldTree` creates a tree based on the breadth-first traversal of a graph. Each time a graph vertex is found, a new tree vertex is created.

Available options:

- `DirectedEdges → False` will ignore edge directions in directed graphs. Otherwise, the search is done only along edge directions.

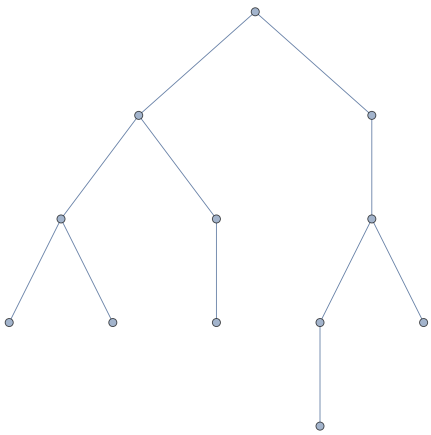
In[651]:=

`tree = IGUnfoldTree[`




`, {1}]`

Out[651]=



The original vertex that generates a tree node is stored in the `"OriginalVertex"` property.

In[652]:=

```
IGVertexProp["OriginalVertex"][tree]
```

Out[652]=

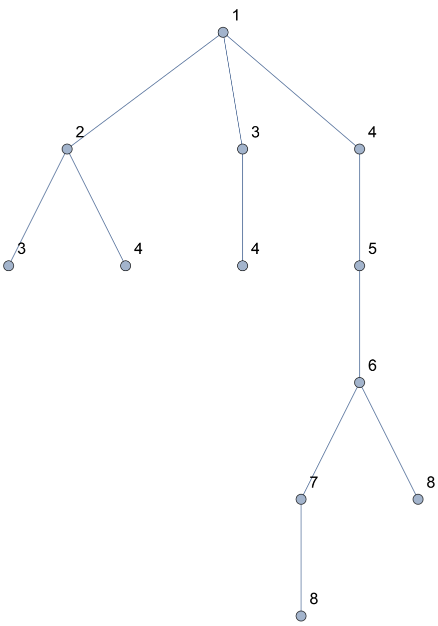
```
{1, 2, 3, 4, 6, 7, 8, 5, 3, 4, 4, 8}
```

We can label the tree nodes with the name of the original vertex either using pattern matching in `VertexLabels` along with `PropertyValue` ...

In[653]:=

```
IGLayoutReingoldTilford[
 tree,
 "RootVertices" → {1}, VertexLabels → (v_ :→ PropertyValue[{tree, v}, "OriginalVertex"])
]
```

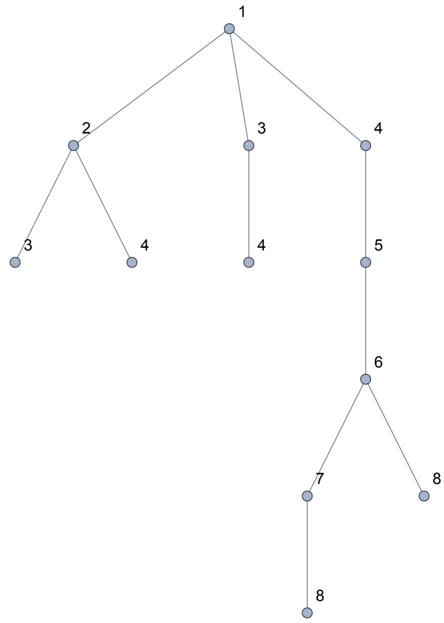
Out[653]=



… or using `IGVertexMap`.

In[654]:=
```
IGLayoutReingoldTilford[tree, "RootVertices" → {1}] //
 IGVertexMap[# &, VertexLabels → IGVertexProp["OriginalVertex"]]
```

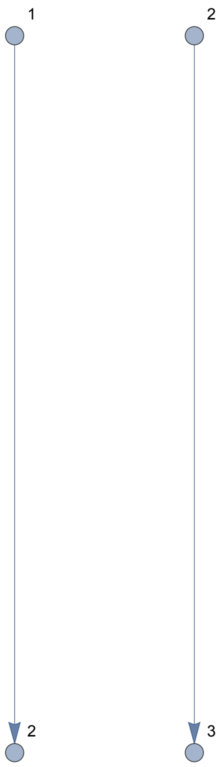
Out[654]=




In directed graphs, the search is done along edge directions. It may be necessary to give multiple starting roots to fully unfold a weakly connected (or unconnected) graph.

In[655]:=
```
IGUnfoldTree[Graph[{1 → 2, 2 → 3}], {2, 1}] //
 IGVertexMap[# &, VertexLabels → IGVertexProp["OriginalVertex"]]
```

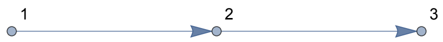
Out[655]=



Use `DirectedEdges → False` to ignore edge directions during the search. Edge directions are still preserved in the result.

In[656]:=
```
IGUnfoldTree[Graph[{1 → 2, 2 → 3}], {2}, DirectedEdges → False] //
 IGVertexMap[# &, VertexLabels → IGVertexProp["OriginalVertex"]]
```

Out[656]=




## Other structural properties

### IGNullGraphQ

In[657]:=
```
? IGNullGraphQ
```

IGNullGraphQ[graph] tests whether graph has no vertices.

`IGNullGraphQ` returns `True` only for the null graph, i.e. the graph that has no vertices.

In[658]:=
```
IGNullGraphQ[IGEmptyGraph[]]
```

Out[658]=
```
True
```

For graphs that have vertices, but no edges, it returns `False`.

In[659]:=
```
IGNullGraphQ[IGEmptyGraph[5]]
```

Out[659]=
```
False
```

In contrast, the built-in `EmptyGraphQ` tests if there are no edges:

In[660]:=
```
EmptyGraphQ[IGEmptyGraph[5]]
```

Out[660]=
```
True
```

### IGCompleteQ

In[661]:=
```
? IGCompleteQ
```

IGCompleteQ[graph] tests if all pairs of vertices are connected in graph.

`IGCompleteQ` tests if a graph is complete, i.e. if all pairs of vertices are connected.

In[662]:=
```
IGCompleteQ@IGCompleteGraph[10]
```

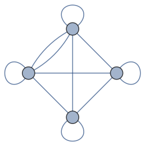
Out[662]=
```
True
```

In[663]:=
```
IGCompleteQ@IGCompleteGraph[5, DirectedEdges → True]
```

Out[663]=
```
True
```

`IGCompleteQ` ignores self-loops and multi-edges.

In[664]:=

```
IGCompleteQ[
```


```
]
```

Out[664]=

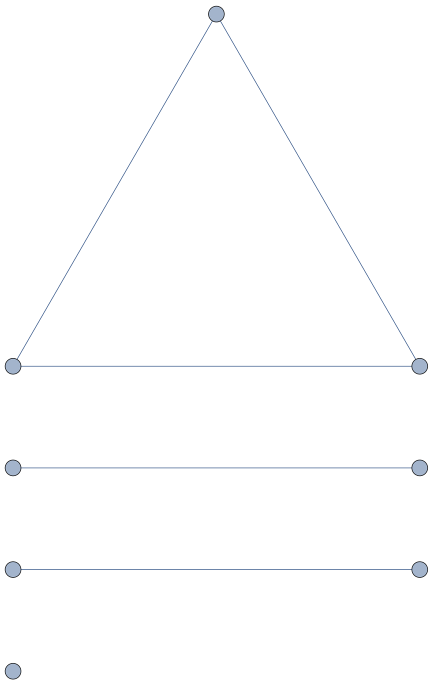
```
True
```

Check if each connected component of a graph is a clique.

In[665]:=

```
g = GraphData[{8, 911}]
```

Out[665]=



In[666]:=

```
AllTrue[ConnectedGraphComponents[g], IGCompleteQ]
```

Out[666]=

```
True
```

The null graph is considered complete.

In[667]:=

```
IGCompleteQ@IGEmptyGraph[]
```

Out[667]=

```
True
```

## IGCactusQ

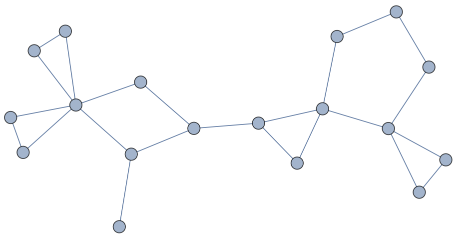
In[668]:=

```
? IGCactusQ
```

IGCactusQ[graph] tests if graph is a cactus

`IGCactusQ` tests if a graph is a cactus. A cactus graph is a connected undirected graph in which any two simple cycles share at most one vertex. Equivalently, a cactus is a connected graph in which every edge belongs to at most one simple cycle.

In[669]:=

IGCactusQ[  ]

Out[669]=

```
True
```

In[670]:=

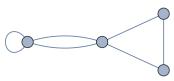
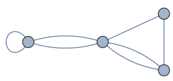
```
IGCactusQ[GridGraph[{2, 3}]]
```

Out[670]=

```
False
```

`IGCactusQ` supports multigraphs and ignores self-loops.

In[671]:=

IGCactusQ /@ {  ,  }

Out[671]=

```
{True, False}
```

The null graph is not considered to be a cactus, but the singleton graph is.

In[672]:=

```
IGCactusQ /@ {IGEmptyGraph[0], IGEmptyGraph[1]}
```

Out[672]=

```
{False, True}
```

Currently, `IGCactusQ` does not support directed graphs.

In[673]:=

```
IGCactusQ[Graph[{1 → 2}]]
```

IGCactusQ: IGCactusQ is not implemented for directed graphs.

Out[673]=

```
$Failed
```

# Motifs and subgraphs

## Motifs

IGraph/M’s motif-related functions count the number of times each possible connectivity pattern of *k* vertices (i.e. induced subgraph of size *k*) occurs in a graph. The patterns are called *motifs*. As of IGraph/M 0.6, size 3 and 4 motifs are supported in directed graphs and size 3 to 6 in undirected graphs. Only (weakly) connected subgraphs are considered.

To count larger induced subgraphs, see `IGLADSubisomorphismCount`. To identify where a subgraph occurs, see `IGLADFindSubisomorphisms`.

To count non-connected size-3 subgraphs, use `IGTriadCensus`.

igraph’s motif functions use the RAND-ESU algorithm, which is able to uniformly sample a random subset of motifs (connected subgraphs), and can thus estimate motif counts even in very large graphs. See the description of `IGMotifs` for an example.

### References


- S. Wernicke, *Efficient Detection of Network Motifs*, IEEE/ACM Trans. Comput. Biol. Bioinforma. **3**, 347 (2006).


## IGMotifs

In[674]:=

```
? IGMotifs
```

IGMotifs[graph, motifSize] gives the motif distribution of graph. See IGIsoclass and IGData for motif ordering.
IGMotifs[graph, motifSize, cutProbabilities] terminates the search with the given probability at each level of the ESU tree.

`IGMotifs` counts how many times each motif (i.e. induced subgraph) of the given size occurs in the graph. For subgraphs that are not weakly connected, `Indeterminate` is returned.

Available options are:

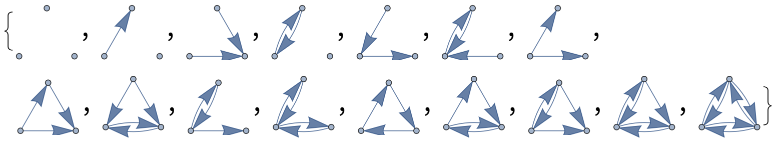
- `DirectedEdges → False` treats the graph as undirected and `DirectedEdges → True` treats the graph as directed. The default is `DirectedEdges → Automatic`, which respects the directedness of the graph.

Motifs are returned by their `IGIsoclass`, i.e. the same order as listed in `IGData`.

In[675]:=

```
mot3 = Graph[#, ImageSize → 36, VertexSize → 0.1] & /@ IGData[{"AllDirectedGraphs", 3}]
```

Out[675]=

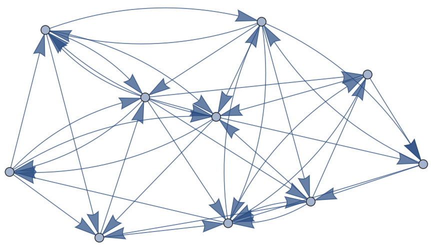


Let us count size-3 motifs in the following graph, and summarize them a table. For non-weakly-connected subgraphs, `Indeterminate` is returned.

In[676]:=

```
g = RandomGraph[{10, 40}, DirectedEdges → True]
```

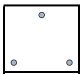
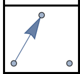
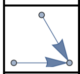
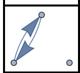
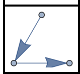
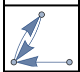
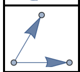
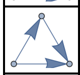
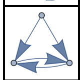
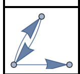
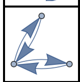
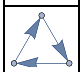
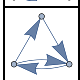
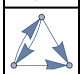
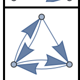
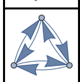
Out[676]=



In[677]:=

```
Grid[{mot3, IGMotifs[g, 3]}ᵀ, Frame → All]
```

Out[677]=

| | |
|---|---|
|  | Indeterminate |
|  | Indeterminate |
|  | 7 |
|  | Indeterminate |
|  | 14 |
|  | 11 |
|  | 3 |
|  | 11 |
|  | 3 |
|  | 16 |
|  | 3 |
|  | 5 |
|  | 7 |
|  | 3 |
|  | 9 |
|  | 0 |

Empty graphs are treated as undirected by default. To treat them as directed, use `DirectedEdges → True`. The result will be different as the number of non-isomorphic graphs on *k* vertices is not the same in the directed and undirected cases.

In[678]:=

```
IGMotifs[IGEmptyGraph[5], 3, DirectedEdges → #] & /@ {Automatic, True, False}
```

Out[678]=

```
{{Indeterminate, Indeterminate, 0, 0},
 {Indeterminate, Indeterminate, 0, Indeterminate, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {Indeterminate, Indeterminate, 0, 0}}
```

### Example: metabolic network

Let us find the size-4 motifs that stand out in the *E. coli* metabolic network by comparing the motif counts to that of a rewired graph:

In[679]:=

```
g = ExampleData[{"NetworkGraph", "MetabolicNetworkEscherichiaColi"}];
```

In[680]:=

```
rg = IGRewire[g, 50 000];
```

In[681]:=

```
ratios = N@Quiet[IGMotifs[g, 4] / IGMotifs[rg, 4]]
```

Out[681]=

```
{Indeterminate, Indeterminate, Indeterminate, 1.30949, Indeterminate, Indeterminate,
 Indeterminate, 1.31896, 0.353667, Indeterminate, Indeterminate, Indeterminate, 0.578637,
 0.58617, 0., Indeterminate, 0.016559, 0., 0., 4.94656, 0., 0., Indeterminate, Indeterminate,
 1.29702, 0.346928, 0.165433, Indeterminate, Indeterminate, 0.674934, 0., 0.0190229,
 0., Indeterminate, Indeterminate, 0., 0., 0., 0., Indeterminate, 0.28928, 0.708281, 0.,
 0.300212, 0., 0.188359, 0.0603697, 0., 0., 0., 0., 0., 1.37343, 0., 0.0753138, 0., 0., 0.,
 0., 0., 0., 0., Indeterminate, 0., 0., 0., 31.1911, 0., 0.21875, 0., 0., 0., 0., 0.186047,
 0., 0., 1.25741, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., Indeterminate,
 0.341903, 0.129316, 1.38161, 0., 0.0207404, 0., 0.245275, 0., 0.046376, 0., 0., 0., 0., 0.,
 0., 0., 0., 0., 0.589124, 0., 0.275862, 0., 0., 0., 0., 0., 0., 0., Indeterminate, 0.429719,
 0., 0., 0., 2.11212, 0., 0., 44.6957, 0., 0.557143, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
 0., 0., 0., 0., 0., 2.22222, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.0769231, 0.,
 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., Indeterminate,
 0., 0., 0., 0., Indeterminate, 0., 0., 0., 0., 0., 0., Indeterminate, 0., Indeterminate}
```

In[682]:=

```
largeRatios = Select[ratios, # > 5 &]
```

Out[682]=

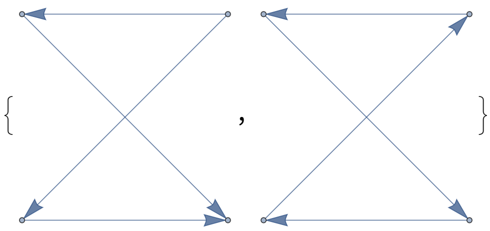
```
{31.1911, 44.6957}
```

There are two motifs that are more than 30 times more common in the metabolic network than in the rewired graph.

In[683]:=

```
Extract[IGData[{"AllDirectedGraphs", 4}], FirstPosition[ratios, #]] & /@ largeRatios
```

Out[683]=

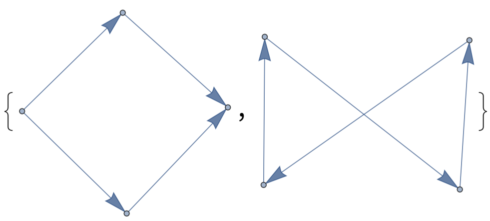


The Davidson–Harel algorithm attempts to reduce edge crossings and can draw these subgraphs in a clearer way:

In[684]:=

```
IGLayoutDavidsonHarel /@ %
```

Out[684]=



### Estimating motif counts in large graphs

`IGMotifs` uses the RAND-ESU algorithm which can uniformly sample a random subset of motifs, and thus estimate motif counts even in very large graphs. To enable random sampling, set a cutoff probability `cutoff = {p1, p2, …, pn}` for stopping the search at each level of the ESU tree. The length of the cutoff probability vector, n, must be the same as the motif size. The number of sampled motifs is, on average, a fraction $(1 - p_1) \times (1 - p_2) \times \ldots \times (1 - p_n)$ of the total number.

In[685]:=

```
bigG = ExampleData[{"NetworkGraph", "WorldWideWeb"}];
{VertexCount[bigG], EdgeCount[bigG]}
```

Out[686]=

```
{325 729, 1 497 134}
```

Sample a fraction $0.1^3 = 0.001$ of all motifs.

In[687]:=

```
IGMotifs[bigG, 3, 1 - 0.1 {1, 1, 1}] // AbsoluteTiming
```

Out[687]=

```
{0.676143, {Indeterminate, Indeterminate, 34 953,
  Indeterminate, 1549, 1646, 27 314, 378, 291, 681, 3917, 2, 49, 171, 71, 7010}}
```

Sample 12.5% of motifs, i.e. a fraction of $0.5^3$.

In[688]:=

```
IGMotifs[bigG, 3, 1 - 0.5 {1, 1, 1}] // AbsoluteTiming
```

Out[688]=

```
{11.8555, {Indeterminate, Indeterminate, 36 350 526, Indeterminate, 299 314,
  530 929, 5 424 564, 67 262, 34 048, 166 963, 516 326, 1730, 4461, 204 276, 12 329, 800 823}}
```

## IGMotifsVertexParticipation

In[689]:=

```
? IGMotifsVertexParticipation
```

IGMotifsVertexParticipation [graph, motifSize] counts the number of times each vertex occurs in each motif.

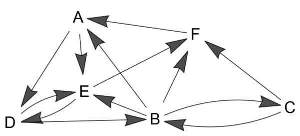
`IGMotifsVertexParticipation` counts how many times each vertex participates in each motif. For each vertex, the result is returned in the same format as with `IGMotifs`.

Available options are:

- `DirectedEdges → False` treats the graph as undirected and `DirectedEdges → True` treats the graph as directed. The default is `DirectedEdges → Automatic`, which respects the directedness of the graph.

Count how many times each vertex appears in each 3-motif in a directed graph.

In[690]:=

g = 

;

In[691]:=

```
mot = IGMotifsVertexParticipation[g, 3]
```

Out[691]=

```
<|A → {Indeterminate, Indeterminate, 0, Indeterminate, 2, 0, 0, 2, 1, 1, 0, 2, 0, 0, 0, 0},
 B → {Indeterminate, Indeterminate, 0, Indeterminate, 1, 1, 0, 3, 0, 2, 0, 1, 1, 1, 0, 0},
 C → {Indeterminate, Indeterminate, 1, Indeterminate, 1, 1, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0},
 D → {Indeterminate, Indeterminate, 0, Indeterminate, 2, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0},
 E → {Indeterminate, Indeterminate, 1, Indeterminate, 0, 0, 0, 2, 1, 2, 0, 1, 1, 0, 0, 0},
 F → {Indeterminate, Indeterminate, 1, Indeterminate, 3, 0, 0, 2, 0, 1, 0, 1, 0, 1, 0, 0}|>
```

The sum of the participation counts in 3-motifs is 3 times the total motif counts of the graph.

In[692]:=
```
Total[mot] === 3 IGMotifs[g, 3]
```
Out[692]=
```
True
```

## IGMotifsTotalCount and IGMotifsTotalCountEstimate

In[693]:=
```
? IGMotifsTotalCount
```

IGMotifsTotalCount [graph, motifSize] gives the total
count of motifs (weakly connected subgraphs) of the given size in the graph.
IGMotifsTotalCount [graph, motifSize, cutProbabilities]
terminates the search with the given probability at each level of the ESU tree.

In[694]:=
```
? IGMotifsTotalCountEstimate
```

IGMotifsTotalCountEstimate [graph, motifSize, sampleSize] estimates the total count of motifs
(weakly connected subgraphs) of the given size in graph, based on a vertex sample of the given size.
IGMotifsTotalCountEstimate [graph, motifSize, vertices] uses the specified vertices as the sample.
IGMotifsTotalCountEstimate [graph, motifSize, sample,
cutProbabilities] terminates the search with the given probability at each level of the ESU tree.

`IGMotifsTotalCount[graph, motifSize]` counts the number of weakly connected subgraphs of the given size in a graph. All subgraph sizes greater than 2 are supported.

`IGMotifsTotalCountEstimate[graph, motifSize, sampleSize]` estimates the total number of motifs by taking a random subset of vertices of the specified size, and counting motifs in which these vertices participate. The total number is estimated as `motifCount × vertexCount / sampleSize`.
`IGMotifsTotalCountEstimate[graph, motifSize, vertices]` uses the specified vertices as the sample.

Let us create a graph.

In[695]:=
```
g = RandomGraph[{20, 50}];
```

The number of size-4 subgraphs it has is:

In[696]:=
```
Binomial[VertexCount[g], 4]
```
Out[696]=
```
4845
```

However, only a small fraction of these is connected:

In[697]:=
```
IGMotifsTotalCount[g, 4]
```
Out[697]=
```
779
```

`IGMotifsTotalCount` is effectively equivalent to (but much faster than) the following:

In[698]:=
```
Count[Subsets[VertexList[g], {4}], subset_ /; WeaklyConnectedGraphQ@Subgraph[g, subset]]
```
Out[698]=
```
779
```

Estimate the count of connected subgraphs by subsampling: at each level of the ESU tree, continue only with probability 0.9.

In[699]:=

```
IGMotifsTotalCount[g, 4, 1 - 0.9 {1, 1, 1, 1}] / 0.9^4
```

Out[699]=

```
833.714
```

Estimate the count of connected subgraphs by considering a random subset of 15 vertices (out of a total of 20).

In[700]:=

```
IGMotifsTotalCountEstimate[g, 4, 15]
```

Out[700]=

```
994
```

Use the first 15 vertices tot estimate the count.

In[701]:=

```
IGMotifsTotalCountEstimate[g, 4, Range[15]]
```

Out[701]=

```
1038
```

## Triad and dyad census

In[702]:=

```
? IGTriadCensus
```

IGTriadCensus[graph] classifies triads in the graph
into 16 possible states, labelled using MAN (mutual, asymmetric, null) notation.

In[703]:=

```
? IGDyadCensus
```

IGDyadCensus[graph] classifies dyad in the graph into mutual, asymmetric or null states.

See `IGData["MANTriadLabels"]` for the mapping between MAN labels and graphs.

`IGTriadCensus[g]` does not return triad counts in the same order as `IGMotifs[g, 3]`, i.e. ordered according to the triads' `IGIsoclass[]`. To get the result ordered by isoclass, use

```
Lookup[IGTriadCensus[g], Keys@IGData["MANTriadLabels"]]
```

`IGData["MANTriadLabels"]` are ordered according to isoclass.

In[704]:=

```
net = ExampleData[{"NetworkGraph", "MetabolicNetworkActinobacillusActinomycetemcomitans"}];
```

In[705]:=

```
IGDyadCensus[net]
```

Out[705]=

```
<|Mutual → 32, Asymmetric → 2304, Null → 490 192|>
```

In[706]:=

```
IGTriadCensus[net]
```

Out[706]=

```
<|003 → 160 429 739, 012 → 2 191 799, 102 → 30 579, 021D → 11 774, 021U → 10 566, 021C → 22 853, 111D → 496,
 111U → 583, 030T → 0, 030C → 0, 201 → 27, 120D → 0, 120U → 0, 120C → 0, 210 → 0, 300 → 0|>
```

# Finding triangles

## IGTriangles

In[707]:=

```
? IGTriangles
```

IGTriangles[graph] lists all triangles in the graph. Edge directions are ignored.

Highlight all triangles in a graph.

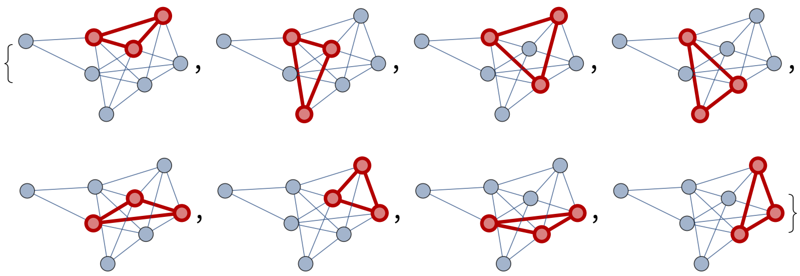
In[708]:=

```
g = RandomGraph[{8, 16}, VertexSize → Large];
```

In[709]:=

```
HighlightGraph[g, Subgraph[g, #], ImageSize → Tiny, GraphHighlightStyle → "Thick"] & /@
 IGTriangles[g]
```

Out[709]=



## IGAdjacenctTriangleCount

In[710]:=

```
? IGAdjacentTriangleCount
```

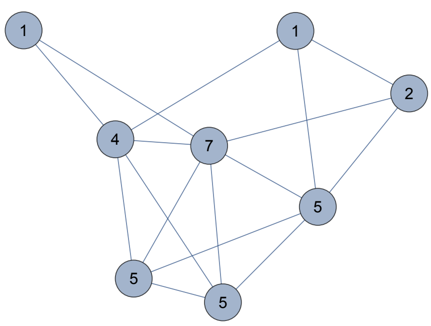
IGAdjacentTriangleCount[graph] counts the triangles each vertex participates in. Edge directions are ignored.
IGAdjacentTriangleCount[graph, vertex] counts the triangles vertex participates in.
IGAdjacentTriangleCount[graph, {vertex1, vertex2, …}] counts the triangles the specified vertices participate in.

Label a graph's vertices based on the number of adjacent triangles.

In[711]:=

```
RandomGraph[{8, 16}, VertexSize → Large] //
 IGVertexMap[Placed[#, Center] &, VertexLabels → IGAdjacentTriangleCount]
```

Out[711]=




## IGTriangleFreeQ

Triangle-free graphs do not have any fully connected subgraphs of size 3. Equivalently, they do not have any cliques (other than 2-cliques, which are edges).

In[712]:=
```
? IGTriangleFreeQ
```

IGTriangleFreeQ[graph] tests if graph is triangle–free.

Mycielski graphs are triangle-free.

In[713]:=
```
IGTriangleFreeQ@GraphData[{"Mycielski", 10}]
```

Out[713]=
```
True
```

# Isomorphism and the automorphism group

igraph implements three isomorphism testing algorithms: BLISS, VF2 and LAD. These support slightly different functionality.

**Naming:** Most of IGraph/M's isomorphism related functions include the name of the algorithm as a prefix, e.g. `IGBlissIsomorphicQ`. Functions named as `…GetIsomorphism` will find a single isomorphism. Functions named as `…FindIsomorphisms` can find multiple isomorphisms. Both return a result in a format compatible with the built-in `FindGraphIsomorphism`.

Additionally, `IGIsomorphicQ[]` and `IGSubisomorphicQ[]` try to select the best algorithm for the given graphs. For graphs without multi-edges, they use igraph’s default algorithm selection. For multigraphs, they use VF2 after internally transforming the multigraphs to edge- and vertex-coloured simple graphs, in a manner similar to `IGColoredSimpleGraph`.

## Basic functions

### IGIsomorphicQ

In[714]:=
```
? IGIsomorphicQ
```

IGIsomorphicQ[graph1, graph2] tests if graph1 and graph2 are isomorphic.

In[715]:=
```
? IGGetIsomorphism
```

IGGetIsomorphism[graph1, graph2] gives one isomorphism between graph1 and graph2, if it exists.

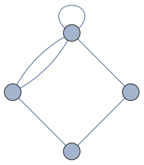
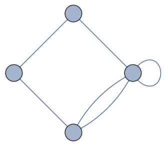
`IGIsomorphicQ` decides if two graphs are isomorphic.

In[716]:=
```
IGIsomorphicQ[IGShorthand["a-b-c-a-d"], IGShorthand["1-2,3-4-2-3"]]
```

Out[716]=
```
True
```

`IGIsomorphicQ` supports multigraphs.

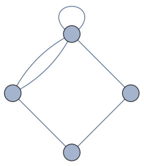
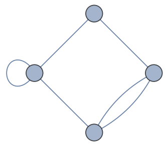
In[717]:=

IGIsomorphicQ[  ,  ]

Out[717]=

```
True
```

In[718]:=

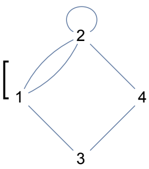
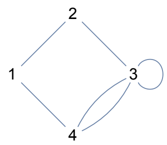
IGIsomorphicQ[  ,  ]

Out[718]=

```
False
```

Get a specific mapping between the vertices of the graphs.

In[719]:=

IGGetIsomorphism[   ,   ]

Out[719]=

```
{<|1 → 4, 2 → 3, 3 → 1, 4 → 2|>}
```

When the graphs are not isomorphic, an empty list is returned.

In[720]:=

```
IGGetIsomorphism[CycleGraph[4], IGCompleteGraph[4]]
```

Out[720]=

```
{}
```

## IGSubisomorphicQ

In[721]:=

```
? IGSubisomorphicQ
```

IGSubisomorphicQ[subgraph, graph] tests if subgraph is contained within graph.

In[722]:=

```
? IGGetSubisomorphism
```

IGGetSubisomorphism[subgraph, graph] gives one subisomorphism from subgraph to graph, if it exists.

`IGSubisomorphicQ` decides if a subgraph is part of a larger graph.

A dodecahedral graph does not contain a `{1, 2, 3}` symmetric tree.

In[723]:=

```
target = GraphData["DodecahedralGraph"];
pattern = IGSymmetricTree[{1, 2, 3}];
```

In[725]:=

```
IGSubisomorphicQ[pattern, target]
```

Out[725]=

```
False
```

It does contain a `{3, 2, 1}` tree.

In[726]:=

```
pattern = IGSymmetricTree[{3, 2, 1}];
IGSubisomorphicQ[pattern, target]
```

Out[727]=

```
True
```

Let us retrieve a specific mapping ...

In[728]:=

```
{iso} = IGGetSubisomorphism[pattern, target]
```

Out[728]=

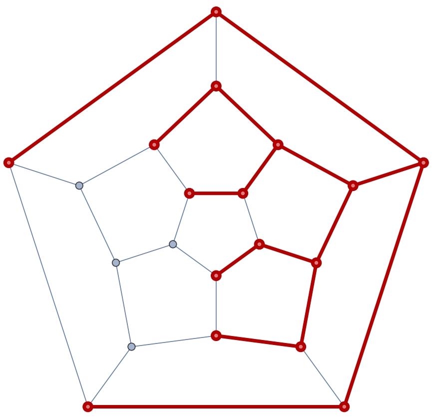
```
{<|1 → 1, 2 → 14, 3 → 15, 4 → 16, 5 → 3, 6 → 9, 7 → 4,
  8 → 10, 9 → 7, 10 → 8, 11 → 19, 12 → 17, 13 → 20, 14 → 18, 15 → 11, 16 → 12|>}
```

... and highlight it.

In[729]:=

```
HighlightGraph[target, VertexReplace[pattern, Normal[iso]],
 GraphHighlightStyle → "Thick"
]
```

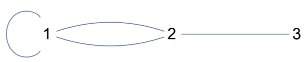
Out[729]=



`IGSubisomorphicQ` supports multigraphs.

In[730]:=

IGSubisomorphicQ[ 

, 

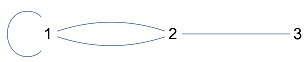
]

Out[730]=

```
True
```

In[731]:=

IGGetSubisomorphism[ 

, 

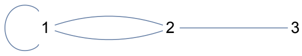
]

Out[731]=

```
{<|a → 1, b → 2|>}
```

In[732]:=

IGSubisomorphicQ[

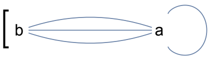
, 

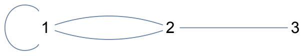
]

Out[732]=

True

In[733]:=

IGSubisomorphicQ[

, 

]

Out[733]=

False

## Bliss

The Bliss library was developed by Tommi Junttila and Petteri Kaski. It is capable of canonical labelling of directed or undirected vertex coloured graphs.

Bliss generally outperforms *Mathematica*'s built-in isomorphisms functions (including finding and counting automorphisms) as of *Mathematica* 12.1. However, this advantage will only be apparent for large and difficult graphs. For small ones the overhead of having to copy the graph and convert it to igraph's internal format is much larger than the actual computation time.

In[734]:=

```
? IGBliss*
```

▼ IGraphM`

| | | |
|---|---|---|
| IGBlissAutomorphismCount | IGBlissCanonicalLabeling | IGBlissIsomorphicQ |
| IGBlissAutomorphismGroup | IGBlissCanonicalPermutation | |
| IGBlissCanonicalGraph | IGBlissGetIsomorphism | |

All Bliss functions take a `"SplittingHeuristics"` option, which can influence the performance of the method. Possible values are:

- `"First"` – First non-unit cell. Very fast but may result in large search spaces on difficult graphs. Use for large but easy graphs.
- `"FirstSmallest"` – First smallest non-unit cell. Fast, should usually produce smaller search spaces than `"First"`.
- `"FirstLargest"` – First largest non-unit cell. Fast, should usually produce smaller search spaces than `"First"`.
- `"FirstMaximallyConnected"` – First maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than `"First"`, `"FirstSmallest"` and `"FirstLargest"`.
- `"FirstSmallestMaximallyConnected"` – First smallest maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than `"First"`, `"FirstSmallest"` and `"FirstLargest"`.
- `"FirstLargestMaximallyConnected"` – First largest maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than `"First"`, `"FirstSmallest"` and `"FirstLargest"`.

The default setting is `"FirstLargest"`, which performs well on average on sparse graphs.

**Note:** The result of the `IGBlissCanonicalLabeling`, `IGBlissCanonicalPermutation` and `IGBlissanonicalGraph` functions depend on the choice of `"SplittingHeuristics"`. See the Bliss documentation for more information.

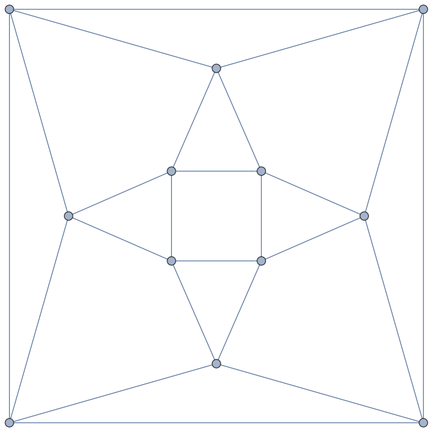
### Basic examples

Let us take the cuboctahedral graph from GraphData …

In[735]:=

```
g1 = GraphData["CuboctahedralGraph"]
```

Out[735]=

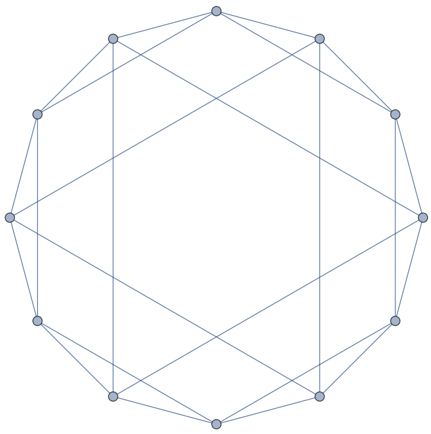


… and also generate it based on its LCF notation.

In[736]:=

```
g2 = IGLCF[{4, 2}, 6]
```

Out[736]=



The two graphs are isomorphic:

In[737]:=

```
IGBlissIsomorphicQ[g1, g2]
```

Out[737]=

```
True
```

One particular mapping between them is the following:

In[738]:=

```
IGBlissGetIsomorphism[g1, g2]
```

Out[738]=

```
{<|1 → 1, 2 → 2, 3 → 5, 4 → 12, 5 → 9, 6 → 4, 7 → 10, 8 → 3, 9 → 6, 10 → 11, 11 → 8, 12 → 7|>}
```

How many mappings are there in total? The same number as the number of automorphisms of either graph.

In[739]:=

```
IGBlissAutomorphismCount[g1]
```

Out[739]=

```
48
```

Bliss cannot generate all 48 of these mappings *directly*. We can either use VF2 for this …

In[740]:=

```
IGVF2FindIsomorphisms[g1, g2] // Length
```

Out[740]=

```
48
```

… or we can use the automorphism group computed by the `IGBlissAutomorphismGroup` function.

In[741]:=

```
group = IGBlissAutomorphismGroup[g1]
```

Out[741]=

```
PermutationGroup[
 {Cycles[{{2, 3}, {4, 5}, {8, 9}, {10, 11}}], Cycles[{{2, 4}, {3, 5}, {6, 7}, {8, 10}, {9, 11}}],
  Cycles[{{1, 2}, {3, 6}, {5, 8}, {7, 10}, {11, 12}}]}]
```

In[742]:=

```
GroupOrder[group]
```

Out[742]=

```
48
```

Ask for all 48 vertex permutations that create isomorphic graphs:

In[743]:=

```
PermutationReplace[VertexList[g1], group]
```

Out[743]=

```
{{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}, {1, 3, 2, 5, 4, 6, 7, 9, 8, 11, 10, 12},
 {1, 4, 5, 2, 3, 7, 6, 10, 11, 8, 9, 12}, {1, 5, 4, 3, 2, 7, 6, 11, 10, 9, 8, 12},
 {2, 1, 6, 4, 8, 3, 10, 5, 9, 7, 12, 11}, {2, 4, 8, 1, 6, 10, 3, 7, 12, 5, 9, 11},
 {2, 6, 1, 8, 4, 3, 10, 9, 5, 12, 7, 11}, {2, 8, 4, 6, 1, 10, 3, 12, 7, 9, 5, 11},
 {3, 1, 6, 5, 9, 2, 11, 4, 8, 7, 12, 10}, {3, 5, 9, 1, 6, 11, 2, 7, 12, 4, 8, 10},
 {3, 6, 1, 9, 5, 2, 11, 8, 4, 12, 7, 10}, {3, 9, 5, 6, 1, 11, 2, 12, 7, 8, 4, 10},
 {4, 1, 7, 2, 10, 5, 8, 3, 11, 6, 12, 9}, {4, 2, 10, 1, 7, 8, 5, 6, 12, 3, 11, 9},
 {4, 7, 1, 10, 2, 5, 8, 11, 3, 12, 6, 9}, {4, 10, 2, 7, 1, 8, 5, 12, 6, 11, 3, 9},
 {5, 1, 7, 3, 11, 4, 9, 2, 10, 6, 12, 8}, {5, 3, 11, 1, 7, 9, 4, 6, 12, 2, 10, 8},
 {5, 7, 1, 11, 3, 4, 9, 10, 2, 12, 6, 8}, {5, 11, 3, 7, 1, 9, 4, 12, 6, 10, 2, 8},
 {6, 2, 3, 8, 9, 1, 12, 4, 5, 10, 11, 7}, {6, 3, 2, 9, 8, 1, 12, 5, 4, 11, 10, 7},
 {6, 8, 9, 2, 3, 12, 1, 10, 11, 4, 5, 7}, {6, 9, 8, 3, 2, 12, 1, 11, 10, 5, 4, 7},
 {7, 4, 5, 10, 11, 1, 12, 2, 3, 8, 9, 6}, {7, 5, 4, 11, 10, 1, 12, 3, 2, 9, 8, 6},
 {7, 10, 11, 4, 5, 12, 1, 8, 9, 2, 3, 6}, {7, 11, 10, 5, 4, 12, 1, 9, 8, 3, 2, 6},
 {8, 2, 10, 6, 12, 4, 9, 1, 7, 3, 11, 5}, {8, 6, 12, 2, 10, 9, 4, 3, 11, 1, 7, 5},
 {8, 10, 2, 12, 6, 4, 9, 7, 1, 11, 3, 5}, {8, 12, 6, 10, 2, 9, 4, 11, 3, 7, 1, 5},
 {9, 3, 11, 6, 12, 5, 8, 1, 7, 2, 10, 4}, {9, 6, 12, 3, 11, 8, 5, 2, 10, 1, 7, 4},
 {9, 11, 3, 12, 6, 5, 8, 7, 1, 10, 2, 4}, {9, 12, 6, 11, 3, 8, 5, 10, 2, 7, 1, 4},
 {10, 4, 8, 7, 12, 2, 11, 1, 6, 5, 9, 3}, {10, 7, 12, 4, 8, 11, 2, 5, 9, 1, 6, 3},
 {10, 8, 4, 12, 7, 2, 11, 6, 1, 9, 5, 3}, {10, 12, 7, 8, 4, 11, 2, 9, 5, 6, 1, 3},
 {11, 5, 9, 7, 12, 3, 10, 1, 6, 4, 8, 2}, {11, 7, 12, 5, 9, 10, 3, 4, 8, 1, 6, 2},
 {11, 9, 5, 12, 7, 3, 10, 6, 1, 8, 4, 2}, {11, 12, 7, 9, 5, 10, 3, 8, 4, 6, 1, 2},
 {12, 8, 9, 10, 11, 6, 7, 2, 3, 4, 5, 1}, {12, 9, 8, 11, 10, 6, 7, 3, 2, 5, 4, 1},
 {12, 10, 11, 8, 9, 7, 6, 4, 5, 2, 3, 1}, {12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}}
```

Permuting the adjacency matrix with any of these leaves it invariant.

In[744]:=

```
perms = PermutationList[#, VertexCount[g1]] & /@ GroupElements[group];
Equal @@ (AdjacencyMatrix[g1][[#, #]] & /@ perms)
```

Out[745]=

```
True
```

Bliss works by computing a canonical labelling of vertices. Then isomorphism can be tested for by comparing the canonically relabelled graphs.

In[746]:=
```
IGBlissCanonicalGraph[g1] === IGBlissCanonicalGraph[g2]
```
Out[746]=
```
True
```

`IGBlissCanonicalGraph` returns graphs in a consistent format so that two graphs are isomorphic if and only if their canonical graphs will compare equal with `===`. Note that in *Mathematica*, graphs may not always compare equal even if they have the same vertex and edge lists.

The corresponding permutation and labelling are

In[747]:=
```
IGBlissCanonicalPermutation[g1]
```
Out[747]=
```
{12, 11, 9, 10, 8, 7, 6, 5, 3, 4, 2, 1}
```

In[748]:=
```
IGBlissCanonicalLabeling[g1]
```
Out[748]=
```
<|1 → 12, 2 → 11, 3 → 9, 4 → 10, 5 → 8, 6 → 7, 7 → 6, 8 → 5, 9 → 3, 10 → 4, 11 → 2, 12 → 1|>
```

Notice that the canonical labelling is simply

In[749]:=
```
AssociationThread[VertexList[g1], IGBlissCanonicalPermutation[g1]]
```
Out[749]=
```
<|1 → 12, 2 → 11, 3 → 9, 4 → 10, 5 → 8, 6 → 7, 7 → 6, 8 → 5, 9 → 3, 10 → 4, 11 → 2, 12 → 1|>
```

Also notice that it is a mapping from `g1` to `IGBlissCanonicalGraph[g1]`:

In[750]:=
```
MemberQ[
 IGVF2FindIsomorphisms[g1, IGBlissCanonicalGraph[g1]],
 IGBlissCanonicalLabeling[g1]
]
```
Out[750]=
```
True
```

The canonical graph returned by `IGBlissCanonicalGraph` always has vertices labelled by the integers `1, 2,` … It can also be used to filter duplicates from a list of graphs

For example, let us generate all possible adjacency matrices of 3-vertex simple directed graphs.

In[751]:=
```
(* fills nondiagonal entries of n by n matrix from vector *)
toMat[vec_, n_] := SparseArray@Partition[Flatten@Riffle[Partition[vec, n], 0, {1, -1, 2}], n]
```

There are $2^{3\times 2} = 2^6 = 64$ such matrices.

In[752]:=
```
graphs = AdjacencyGraph[toMat[#, 3], DirectedEdges → True] & /@ IntegerDigits[Range[2^6] - 1, 2, 6];
```

But only 16 of them correspond to non-isomorphic graphs

In[753]:=
```
DeleteDuplicatesBy[graphs, IGBlissCanonicalGraph] // Length
```
Out[753]=
```
16
```

When `IGBlissCanonicalGraph` is given a vertex coloured graph, it will encode the colours into a vertex property named `"Color"`. This allows distinguishing between graphs whose canonical graphs are identical in structure, but differ in colouring.

Take for example the following coloured graphs:

In[754]:=

```
g = Graph[{1 ↔ 2, 2 ↔ 3}, VertexSize → Large, GraphStyle → "BasicBlack"];
colg1 = Graph[g, Properties → {1 → {"color" → 1}, 2 → {"color" → 3}, 3 → {"color" → 2}}];
colg2 = Graph[g, Properties → {1 → {"color" → 1}, 2 → {"color" → 3}, 3 → {"color" → 1}}];
```

Visualize them for clarity:

In[757]:=

```
IGVertexMap[ColorData[97], VertexStyle → IGVertexProp["color"]] /@ {colg1, colg2}
```

Out[757]=

The vertex and edge lists of their canonical graphs are identical:

In[758]:=

```
cang1 = IGBlissCanonicalGraph[{colg1, "VertexColors" → "color"}];
cang2 = IGBlissCanonicalGraph[{colg2, "VertexColors" → "color"}];
```

In[760]:=

```
VertexList /@ {cang1, cang2}
EdgeList /@ {cang1, cang2}
```

Out[760]=

```
{{1, 2, 3}, {1, 2, 3}}
```

Out[761]=

```
{{1 ↔ 3, 2 ↔ 3}, {1 ↔ 3, 2 ↔ 3}}
```

But they differ in colouring, and therefore do not compare equal:

In[762]:=

```
IGVertexPropertyList[cang1]
```

Out[762]=

```
{Color, VertexCoordinates, VertexShape, VertexShapeFunction, VertexSize, VertexStyle}
```

In[763]:=

```
IGVertexProp["Color"] /@ {cang1, cang2}
```

Out[763]=

```
{{1, 2, 3}, {1, 1, 3}}
```

In[764]:=

```
cang1 === cang2
```

Out[764]=

```
False
```

The performance of Bliss functions may depend significantly on the choice of splitting heuristics.

In[765]:=

```
g = LineGraph@GraphData[{"Hadamard", {24, 6}}];
timings = {#, First@Timing@IGBlissAutomorphismGroup[g, "SplittingHeuristics" → #]} & /@
   {"First", "FirstSmallest", "FirstLargest", "FirstMaximallyConnected",
    "FirstSmallestMaximallyConnected", "FirstLargestMaximallyConnected"};
TableForm[timings, TableHeadings → {None, {"Splitting heuristics", "Timing (s)"}}]
```

Out[767]//TableForm=

| Splitting heuristics | Timing (s) |
|---|---|
| First | 2.79602 |
| FirstSmallest | 9.10916 |
| FirstLargest | 1.0737 |
| FirstMaximallyConnected | 4.36106 |
| FirstSmallestMaximallyConnected | 4.37917 |
| FirstLargestMaximallyConnected | 1.0749 |

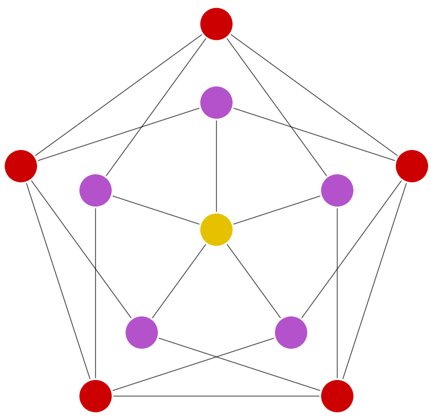
## Additional examples

Let us visualize the vertex equivalence classes induced by a graph's automorphism group. Two vertices are considered equivalent if there is an automorphism that maps one into the other.

In[768]:=

```
With[{g = GraphData[{"Mycielski", 4}]},
 HighlightGraph[g, GroupOrbits@IGBlissAutomorphismGroup[g],
  VertexSize → Large, GraphStyle → "BasicBlack"]
]
```

Out[768]=

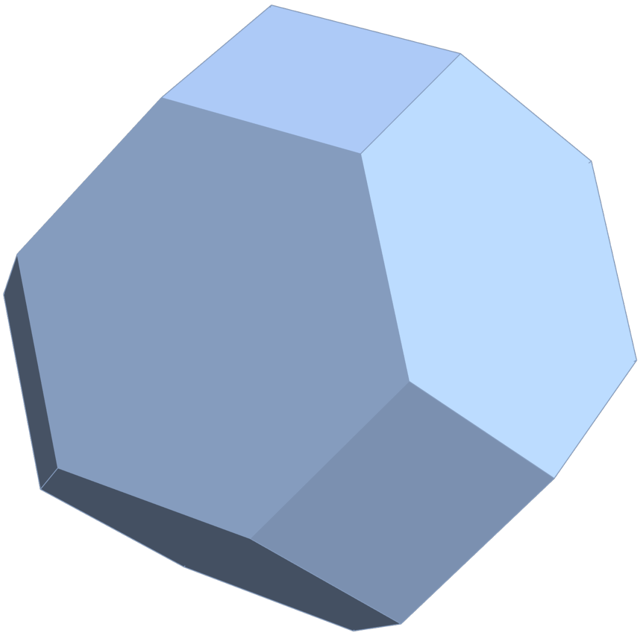


Visualize the edge equivalence classes of a polyhedron, induced by its skeleton's automorphism group.

In[769]:=

```
mesh = PolyhedronData["TruncatedOctahedron", "BoundaryMeshRegion"]
```

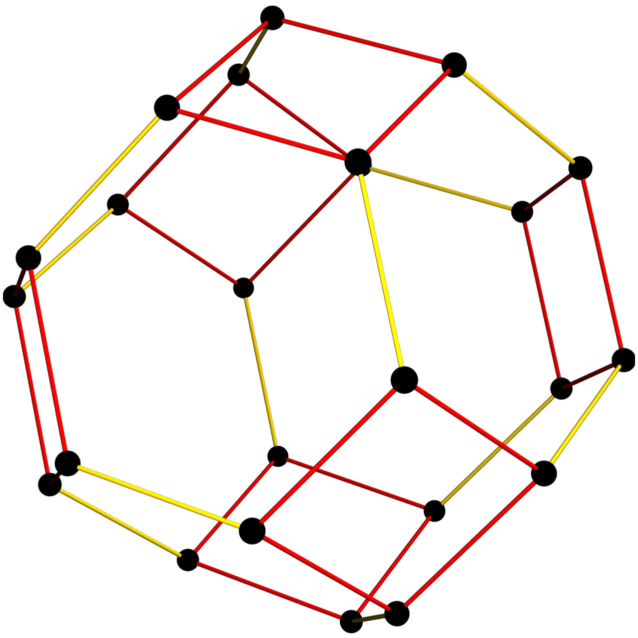
Out[769]=



In[770]:=

```
With[{g = IGMeshGraph[mesh, VertexStyle → Black]},
 HighlightGraph[g,
  EdgeList[g][[#]] & /@ GroupOrbits@IGBlissAutomorphismGroup@LineGraph[g]
 ]
]
```

Out[770]=



## References


- T. Junttila, P. Kaski, Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs, 2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments, doi:10.1137/1.9781611972870.13.

## VF2

In[771]:=

```
? IGVF2*
```

▼IGraphM`

| | | | |
|---|---|---|---|
| IGVF2FindIsomorphisms | IGVF2GetIsomorphism | IGVF2IsomorphicQ | IGVF2SubisomorphicQ |
| IGVF2FindSubisomorphisms | IGVF2GetSubisomorphism | IGVF2IsomorphismCount | IGVF2SubisomorphismCount |

VF2 supports vertex coloured and edge coloured graphs. A colour specification consists of one or more of the `"VertexColors"` and `"EdgeColors"` options. Allowed formats for these options are a list of integers, an association assigning integers to the vertices/edges, or `None`. When using associations, it is not necessarily to specify a colour for each vertex/edge. The omitted ones are assumed to have colour `0`.

The VF2 algorithm only supports simple graphs.

The following graph has two automorphisms: `{1, 2}` and `{2, 1}`.

In[772]:=

```
g = Graph[{1 ↔ 2}];
IGVF2IsomorphismCount[g, g]
```

Out[773]=

```
2
```

If we colour one of the vertices, the permutation `{2, 1}` becomes forbidden, so only one automorphism remains.

In[774]:=

```
IGVF2IsomorphismCount[{g, "VertexColors" → {1, 2}}, {g, "VertexColors" → {1, 2}}]
```

Out[774]=

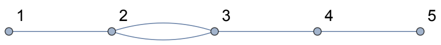
```
1
```

Multigraphs are not directly supported for isomorphism checking, but we can map the multigraph isomorphism problem into an edge-coloured graph isomorphism one by designating the multiplicity of each edge as its colour.

In[775]:=

```
g1 = EdgeAdd[PathGraph[Range[5], VertexLabels → "Name"], 2 ↔ 3]
```

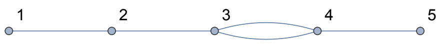
Out[775]=




In[776]:=

```
g2 = EdgeAdd[PathGraph[Range[5], VertexLabels → "Name"], 4 ↔ 3]
```

Out[776]=




In[777]:=

```
IGVF2IsomorphicQ[g1, g2]
```

IGraphM: VF2 does not support non-simple graphs. Consider using IGIsomorphicQ or IGColoredSimpleGraph.

Out[777]=

```
$Failed
```

Since `g1` and `g2` are undirected, we need to bring their edges into a sorted canonical form before counting them. This ensures that 4 ↔ 3 and 3 ↔ 4 are treated as the same edge.

In[778]:=

```
colors1 = Counts[Sort /@ EdgeList[g1]]
```

Out[778]=

```
<|1 ↔ 2 → 1, 2 ↔ 3 → 2, 3 ↔ 4 → 1, 4 ↔ 5 → 1|>
```

In[779]:=

```
colors2 = Counts[Sort /@ EdgeList[g2]]
```

Out[779]=

```
<|1 ⟷ 2 → 1, 2 ⟷ 3 → 1, 3 ⟷ 4 → 2, 4 ⟷ 5 → 1|>
```

In[780]:=

```
IGVF2IsomorphicQ[{Graph@Keys[colors1], "EdgeColors" → colors1},
 {Graph@Keys[colors2], "EdgeColors" → colors2}]
```

Out[780]=

```
True
```

`IGIsomorphicQ` and `IGSubisomorphicQ` check multigraph isomorphism in a similar way, based on edge colouring.

### References

- L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, IEEE Trans. Pattern Anal. Mach. Intell. 26, 1367 (2004).

## LAD

The LAD library was developed by Christine Solnon. It is capable of finding subgraphs in a larger graph.

The LAD algorithm does not support multi-edges.

In[781]:=

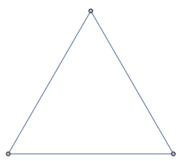
```
? IGLAD*
```

**IGraphM`**

| | | | |
|---|---|---|---|
| IGLADFindSubisomorphisms | IGLADGetSubisomorphism | IGLADSubisomorphicQ | IGLADSubisomorphismCount |

With the `"Induced" → True` option LAD will search for induced subgraphs.

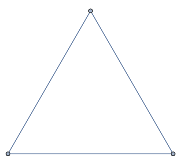
In[782]:=

**IGLADSubisomorphicQ[**  ,  **]**

Out[782]=

```
True
```

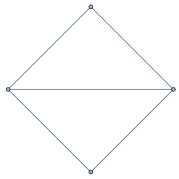
In[783]:=

**IGLADSubisomorphicQ[**  ,  **, "Induced" → True]**

Out[783]=

```
False
```

In[784]:=

**IGLADSubisomorphicQ[**  ,  **, "Induced" → True]**

Out[784]=

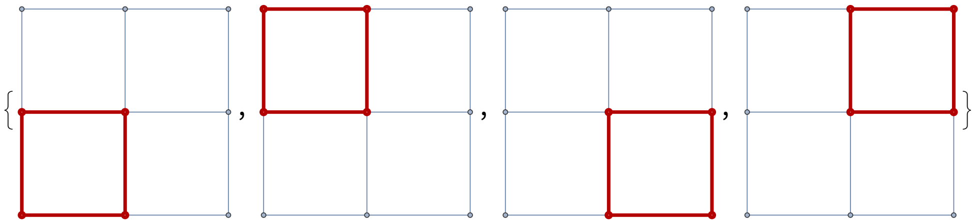
```
True
```

Highlight subgraphs in a grid graph.

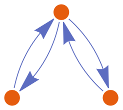
In[785]:=

```
g = GridGraph[{3, 3}];
HighlightGraph[g, Subgraph[g, #], GraphHighlightStyle → "Thick"] & /@
 Union[Sort@*Values /@ IGLADFindSubisomorphisms[GridGraph[{2, 2}], g]]
```

Out[786]=



Count how many times each vertex of a graph appears at the apex of the following subgraph (motif):

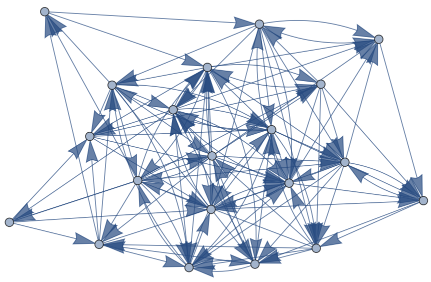


Generate a directed random graph to do the counting in.

In[787]:=

```
g = RandomGraph[{20, 120}, DirectedEdges → True]
```

Out[787]=

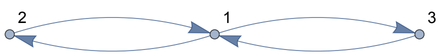


`IGShorthand` provides a concise way to input this subgraph.

In[788]:=

```
motif = IGShorthand["2<->1<->3"]
```

Out[788]=




This motif has a two-fold symmetry, as revealed by its automorphism group. We divide the final counts by two.

In[789]:=

```
Counts@Lookup[
    IGLADFindSubisomorphisms[motif, g, "Induced" → True],
    1
  ] / IGBlissAutomorphismCount[motif]
```

Out[789]=

```
<|13 → 2, 18 → 2, 20 → 4, 12 → 1, 16 → 1|>
```

Check that a graph is claw-free.

In[790]:=

```
clawFreeQ[graph_?UndirectedGraphQ] :=
 Not@IGLADSubisomorphicQ[
   StarGraph[4], (* claw graph *)
   graph,
   "Induced" → True
  ]
```

In[791]:=

```
clawFreeQ /@ {GraphData["DodecahedralGraph"], GraphData["TruncatedPrismGraph"]}
```

Out[791]=

```
{False, True}
```

### References

- Christine Solnon, *AllDifferent*-based filtering for subgraph isomorphism, Artificial Intelligence 174 (2010), doi:10.1016/j.artint.2010.05.002

## Isomorphism of coloured graphs

All three included isomorphism algorithms support vertex coloured graphs, and VF2 supports edge coloured graphs as well. A coloured graph is specified as `{g, "VertexColors" → …, "EdgeColors" → …}`, where both vertex and edge colour specifications are optional. Colours are represented by integers and may be specified in one of the following ways:

- A list of integers, given in the same order as `VertexList[g]` (or `EdgeList[g]` if specifying edge colours).

  `{Graph[{a, b}, {a ↔ b}], "VertexColors" → {1, 2}}`.

- An association assigning integers to vertices (or edges). Vertices (or edges) not present in the association are assumed to have colour `0`.

  `{Graph[{a ↔ b}], "VertexColors" → <|a → 1, b → 2|>}`.

- The name of a a vertex (or edge) property. Vertices (or edges) without an assigned property value are assumed to have colour `0`.

  `{Graph[{Property[a, "color" → 1], Property[b, "color" → 2]}, {a ↔ b}], "VertexColors" → "color"}`

- `"VertexColors" → None` indicates no colouring.

**Example.** Define a graph along with the colours of its vertices.

In[792]:=

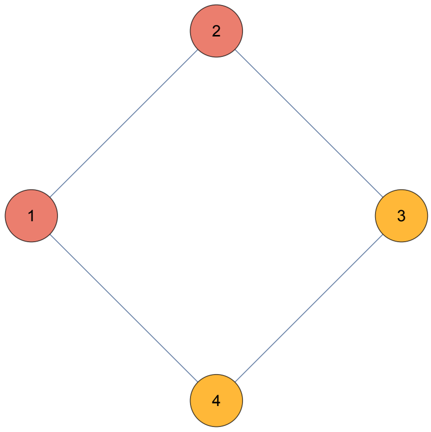
```
g = CycleGraph[4];
vcols = <|
   1 → 1, 2 → 1,
   3 → 2, 4 → 2
  |>;
```

Visualize it.

In[794]:=

```
Graph[g,
 VertexStyle → Normal[ColorData[24] /@ vcols],
 VertexSize → Medium, VertexLabels → Placed["Name", Center]
]
```

Out[794]=




Compute its automorphism group, taking vertex colours into account.

In[795]:=

```
IGBlissAutomorphismGroup[{g, "VertexColors" → vcols}]
```

Out[795]=

```
PermutationGroup[{Cycles[{{1, 2}, {3, 4}}]}]
```

## Properties related to the automorphism group

The functions in this section test for properties related to a graph’s automorphism group. The summary table below illustrates the functions on a set of graphs which all have different properties.

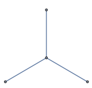
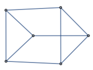
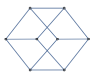
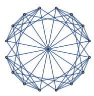
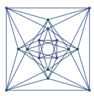
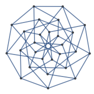
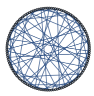
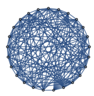
In[796]:=

```
graphs = {StarGraph[4], IGSquareLattice[{2, 3}, "Periodic" → True],
   HypercubeGraph[3], GraphData[{"Rook", {4, 4}}], GraphData["ShrikhandeGraph"],
   GraphData["HoltGraph"], GraphData["Tutte12Cage"], GraphData[{"Paulus", {25, 1}}]};

functions = <|
   "regular" → IGRegularQ,
   "strongly regular" → IGStronglyRegularQ, "distance regular" → IGDistanceRegularQ,
   "vertex transitive" → IGVertexTransitiveQ,
   "edge transitive" → IGEdgeTransitiveQ,
   "arc transitive" → IGEdgeTransitiveQ@*DirectedGraph,
   "distance transitive" → IGDistanceTransitiveQ
  |>;

TableForm[
  Through[Values[functions][#]] & /@ graphs,
  TableHeadings → {Show[#, ImageSize → 50] & /@ graphs, Keys[functions]},
  TableDirections → Row
 ] // Style[#, "Text"] &
```

Out[798]=

| |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| regular | False | True | True | True | True | True | True | True |
| strongly regular | False | False | False | True | True | False | False | True |
| distance regular | False | False | True | True | True | False | True | True |
| vertex transitive | False | True | True | True | True | True | False | False |
| edge transitive | True | False | True | True | True | True | True | False |
| arc transitive | False | False | True | True | True | False | False | False |
| distance transitive | False | False | True | True | False | False | False | False |

### IGRegularQ

In[799]:=

```
? IGRegularQ
```

IGRegularQ[graph] tests if graph is regular, i.e. all vertices have the same degree.
IGRegularQ[graph, k] tests if graph is k–regular, i.e. all vertices have degree k.

`IGRegularQ` checks if a graph is regular. All vertices of a regular graph have the same degrees. In regular directed graphs, the in- and out-degrees are also equal to each other.

In[800]:=

```
IGRegularQ[IGSquareLattice[{3, 4}, "Periodic" → True]]
```

Out[800]=

```
True
```

Check if a graph is $k$-regular for $k = 2$ and $k = 3$.

In[801]:=

```
IGRegularQ[CycleGraph[10], 2]
```

Out[801]=

```
True
```

In[802]:=

```
IGRegularQ[CycleGraph[10], 3]
```

Out[802]=

```
False
```

The null graph is considered 0-regular.

In[803]:=

```
IGRegularQ[IGEmptyGraph[]]
```

Out[803]=

```
True
```

Check if a directed graph is regular.

In[804]:=

```
IGRegularQ[CycleGraph[5, DirectedEdges → True]]
```

Out[804]=

```
True
```

`IGRegularQ` considers self-loops and multi-edges when computing vertex degrees.

In[805]:=

IGRegularQ[  ]

Out[805]=

```
True
```

## IGStronglyRegularQ

In[806]:=

```
? IGStronglyRegularQ
```

IGStronglyRegularQ[graph] tests if graph is strongly regular.

`IGStronglyRegularQ` checks if a graph is strongly regular. A strongly regular graph is a regular graph where each pair of connected vertices have the same number of common neighbours, $\lambda$, and each pair of unconnected vertices also have the same number of common neighbours, $\mu$.

In[807]:=

```
IGStronglyRegularQ@GraphData["ShrikhandeGraph"]
```

Out[807]=

```
True
```

Hypercube graphs and 3 and higher dimensions are not strongly regular, even though they are regular.

In[808]:=
```
IGStronglyRegularQ /@ HypercubeGraph /@ Range[2, 4]
```
Out[808]=
```
{True, False, False}
```

Some authors exclude empty and complete graph from the definition, as they satisfy these conditions trivially. `IGStronglyRegularQ` returns `True` for these.

In[809]:=
```
IGStronglyRegularQ /@ {IGEmptyGraph[5], IGCompleteGraph[6]}
```
Out[809]=
```
{True, True}
```

It also returns `True` for graphs on 0, 1 and 2 vertices.

In[810]:=
```
IGStronglyRegularQ /@ IGCompleteGraph /@ Range[0, 2]
```
Out[810]=
```
{True, True, True}
```

Currently, `IGStronglyRegularQ` does not support directed graphs.

In[811]:=
```
IGStronglyRegularQ@Graph[{1 → 2}]
```
IGStronglyRegularQ : Directed graphs are not supported.

Out[811]=
```
$Failed
```

## IGStronglyRegularParameters

In[812]:=
```
? IGStronglyRegularParameters
```

IGStronglyRegularParameters[graph] returns the parameters {v, k, $\lambda$, $\mu$} of a strongly regular graph. For non–strongly–regular graphs {} is returned.

`IGStronglyRegularParameters` returns the parameters ($v$, $k$, $\lambda$, $\mu$) of a strongly regular graph. $v$ is the number of vertices, $k$ the degree of the vertices, $\lambda$ the number of common neighbours of connected vertices and $\mu$ the number of common neighbours of unconnected vertices.

In[813]:=
```
IGStronglyRegularParameters[PetersenGraph[]]
```
Out[813]=
```
{10, 3, 0, 1}
```

In[814]:=
```
IGStronglyRegularParameters[CycleGraph[5]]
```
Out[814]=
```
{5, 2, 0, 1}
```

The parameters of a strongly regular graph satisfy the equation $(v - k - 1)\,\mu = k(k - \lambda - 1)$.

In[815]:=
```
{v, k, lambda, mu} = IGStronglyRegularParameters[GraphData[{"Paley", 101}]]
```
Out[815]=
```
{101, 50, 24, 25}
```

In[816]:=

```
(v - k - 1) mu == k (k - lambda - 1)
```

Out[816]=

```
True
```

$\lambda$ and $\mu$ are not well-defined for empty and complete graphs, respectively. In these cases, 0 is returned.

In[817]:=

```
IGStronglyRegularParameters /@ {IGEmptyGraph[5], IGCompleteGraph[6]}
```

Out[817]=

```
{{5, 0, 0, 0}, {6, 5, 4, 0}}
```

For non-strongly-regular graphs, { } is returned.

In[818]:=

```
IGStronglyRegularParameters[HypercubeGraph[3]]
```

Out[818]=

```
{}
```

## IGDistanceRegularQ

In[819]:=

```
? IGDistanceRegularQ
```

IGDistanceRegularQ[graph] tests if graph is distance regular.

`IGDistanceRegularGraph` checks if a graph is distance regular.

In[820]:=

```
IGDistanceRegularQ@HypercubeGraph[5]
```

Out[820]=

```
True
```

In[821]:=

```
IGDistanceRegularQ@IGSquareLattice[{2, 5}, "Periodic" → True]
```

Out[821]=

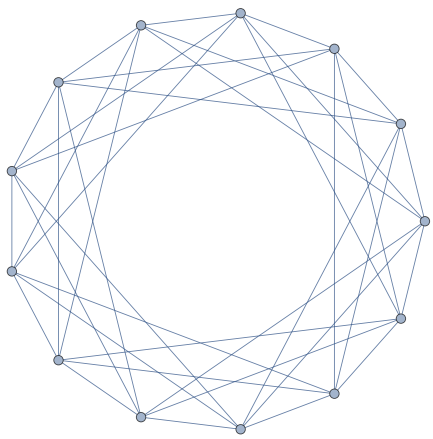
```
False
```

A distance regular graph with a diameter of 2 is also strongly regular.

In[822]:=

```
g = GraphData[{"Paley", 13}]
```

Out[822]=



In[823]:=

```
{IGDiameter[g], IGDistanceRegularQ[g], IGStronglyRegularQ[g]}
```

Out[823]=

```
{2, True, True}
```

The Shrikhande graph is the smallest graph that is distance regular, but not distance transitive.

In[824]:=

```
Through[{IGDistanceRegularQ, IGDistanceTransitiveQ}[GraphData["ShrikhandeGraph"]]]
```

Out[824]=

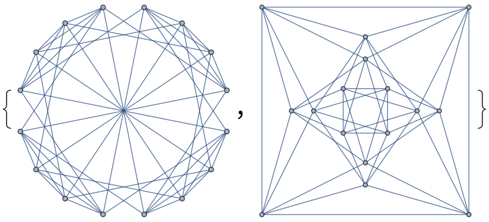
```
{True, False}
```

A disconnected graph is distance regular if its components are distance regular and they are co-spectral. The following graphs are co-spectral:

In[825]:=

```
components = {GraphData[{"Rook", {4, 4}}], GraphData["ShrikhandeGraph"]}
```

Out[825]=



In[826]:=

```
Eigenvalues /@ AdjacencyMatrix /@ components
```

Out[826]=

```
{{6, -2, -2, -2, -2, -2, -2, -2, -2, -2, 2, 2, 2, 2, 2, 2},
 {6, -2, -2, -2, -2, -2, -2, -2, -2, -2, 2, 2, 2, 2, 2, 2}}
```

They are both distance regular with the same intersection array.

In[827]:=

```
IGIntersectionArray /@ components
```

Out[827]=

```
{{{6, 3}, {1, 2}}, {{6, 3}, {1, 2}}}
```

Thus their disjoint union is also distance regular.

In[828]:=

```
IGDistanceRegularQ@IGDisjointUnion[components]
```

Out[828]=

```
True
```

All distance transitive graphs are also distance regular, but the reverse is not true.

In[829]:=

```
IGDistanceTransitiveQ /@ components
```

Out[829]=

```
{True, False}
```

`IGDistanceRegularQ` does not currently support directed graphs or non-simple graphs.

In[830]:=

```
IGDistanceRegularQ[Graph[{1 → 2}]]
```

IGDistanceRegularQ: Directed graphs are not supported.

Out[830]=

```
$Failed
```

In[831]:=

```
IGDistanceRegularQ[Graph[{1 ⟷ 2, 1 ⟷ 2}]]
```

IGDistanceRegularQ: Non–simple graphs are not supported.

Out[831]=

```
$Failed
```

## IGIntersectionArray

In[832]:=

```
? IGIntersectionArray
```

IGIntersectionArray[graph] computes the intersection array {b, c} of a distance regular graph. For non–distance–regular graphs {} is returned.

In[833]:=

```
IGIntersectionArray@GraphData["IcosahedralGraph"]
```

Out[833]=

```
{{5, 2, 1}, {1, 2, 5}}
```

In[834]:=

```
IGIntersectionArray@GraphData["SuzukiGraph"]
```

Out[834]=

```
{{416, 315}, {1, 96}}
```

In[835]:=

```
IGIntersectionArray@CycleGraph[6]
```

Out[835]=

```
{{2, 1, 1}, {1, 1, 2}}
```

For non-distance-regular graphs, `{}` is returned.

In[836]:=

```
IGIntersectionArray[GridGraph[{3, 3}]]
```

Out[836]=

```
{}
```

`IGIntersectionArray` does not currently support directed graphs.

In[837]:=

```
IGIntersectionArray[Graph[{1 → 2}]]
```

IGDistanceRegularQ: Directed graphs are not supported.

Out[837]=

```
$Failed
```

## IGVertexTransitiveQ

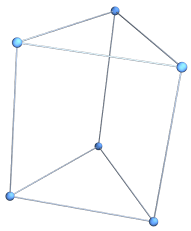
In[838]:=

```
? IGVertexTransitiveQ
```

IGVertexTransitiveQ[graph] tests if graph is vertex transitive.

`IGVertexTransitiveQ` checks if a graph is vertex transitive, i.e. if any vertex can be mapped into any other by some automorphism of the graph.

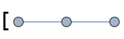
In[839]:=

```
IGVertexTransitiveQ[
```

```
]
```

Out[839]=

```
True
```

In[840]:=

```
IGVertexTransitiveQ[
```

```
]
```

Out[840]=

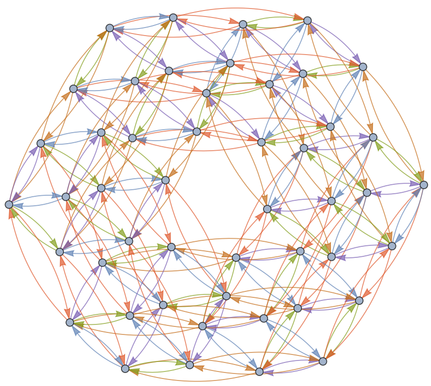
```
False
```

All Cayley graphs are vertex transitive.

In[841]:=

```
cg = CayleyGraph@IGBlissAutomorphismGroup@IGLCF[{2, -1, 2}, 3]
```

Out[841]=



In[842]:=

```
IGVertexTransitiveQ[cg]
```

Out[842]=

```
True
```

## IGEdgeTransitiveQ

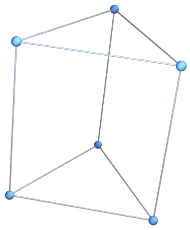
In[843]:=

```
? IGEdgeTransitiveQ
```

IGEdgeTransitiveQ[graph] tests if graph is edge transitive.

`IGEdgeTransitiveQ` checks if a graph is edge transitive, i.e. if any edge can be mapped into any other by some automorphism of the graph.

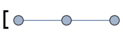
In[844]:=

IGEdgeTransitiveQ[  ]

Out[844]=

```
False
```

In[845]:=

IGEdgeTransitiveQ[  ]

Out[845]=

```
True
```

The Folkman graph is not vertex transitive but it is edge transitive.

In[846]:=

```
Through[{IGVertexTransitiveQ, IGEdgeTransitiveQ}@GraphData["FolkmanGraph"]]
```

Out[846]=

```
{False, True}
```

`IGEdgeTransitiveQ` takes into account edge directions.

In[847]:=

```
IGEdgeTransitiveQ@Graph[{1 → 2, 2 → 3}]
```

Out[847]=

```
False
```

In[848]:=

```
IGEdgeTransitiveQ@Graph[{1 → 2, 3 → 2}]
```

Out[848]=

```
True
```

Arc transitivity in an undirected graph refers to edge transitivity when each undirected edge is replaced by two opposite directed edges.

In[849]:=

```
arcTransitiveQ[graph_?UndirectedGraphQ] := IGEdgeTransitiveQ@DirectedGraph[graph]
```

Some graphs are edge transitive, but not arc transitive.

In[850]:=

```
IGEdgeTransitiveQ@GraphData[{"Bouwer", {2, 4, 15}}]
```

Out[850]=

```
True
```

In[851]:=

```
arcTransitiveQ@GraphData[{"Bouwer", {2, 4, 15}}]
```

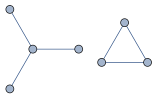
Out[851]=

```
False
```

Most graphs are edge transitive if their line graphs are vertex transitive. The exceptions are disjoint unions of the 3-star and 3-cycle. These two graphs have the same line graph, but they are not isomorphic.

In[852]:=

```
g = 
```



```
;
```

In[853]:=

```
{IGEdgeTransitiveQ[g], IGVertexTransitiveQ@LineGraph[g]}
```

Out[853]=

```
{False, True}
```

## IGSymmetricQ

In[854]:=

```
? IGSymmetricQ
```

IGSymmetricQ[graph] tests if graph is symmetric, i.e. if it is both vertex transitive and edge transitive.

`IGSymmetricQ` checks if a graph is both vertex transitive and edge transitive. Note that this property is distinct from being *arc transitive*, which is the definition used for “*symmetric*” by some authors.

In[855]:=

```
IGSymmetricQ[GraphData["DodecahedralGraph"]]
```

Out[855]=

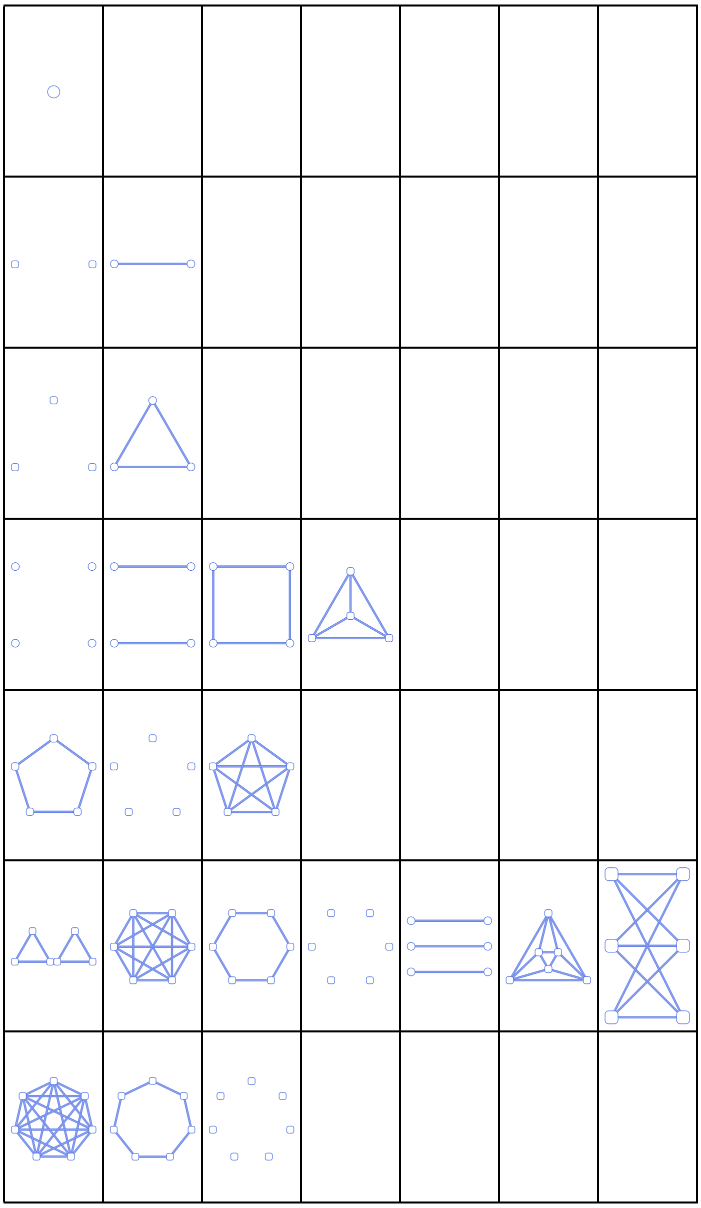
```
True
```

Make a table of symmetric graphs up to size 7:

In[856]:=

```
Grid[
 Table[
  Graph[#, ImageSize → 50, PlotTheme → "Business"] & /@
   Select[GraphData /@ GraphData[k], IGSymmetricQ],
  {k, 1, 7}
 ], Frame → All, ItemSize → All]
```

Out[856]=

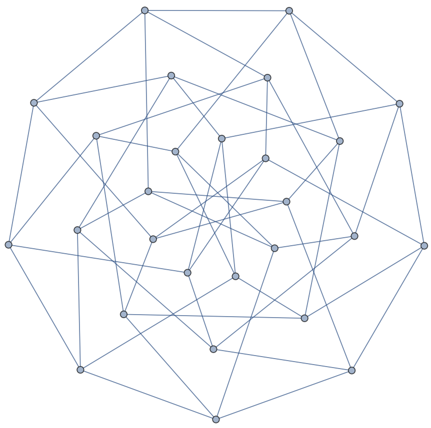


Some authors use the term *symmetric graph* to refer to arc transitive graphs. Arc transitivity can be checked using `IGEdgeTransitiveQ@DirectedGraph[#] &`. All arc-transitive graphs are both vertex- and edge-transitive, but the reverse is not true. The smallest graph that is both vertex- and edge-transitive, but not arc-transitive, is the 27-vertex Doyle graph, also known as the Holt graph.

In[857]:=

```
doyle = GraphData["DoyleGraph"]
```

Out[857]=



In[858]:=

```
{IGVertexTransitiveQ[doyle], IGEdgeTransitiveQ[doyle]}
```

Out[858]=

```
{True, True}
```

In[859]:=

```
IGEdgeTransitiveQ@DirectedGraph[doyle]
```

Out[859]=

```
False
```

## IGDistanceTransitiveQ

In[860]:=

```
? IGDistanceTransitiveQ
```

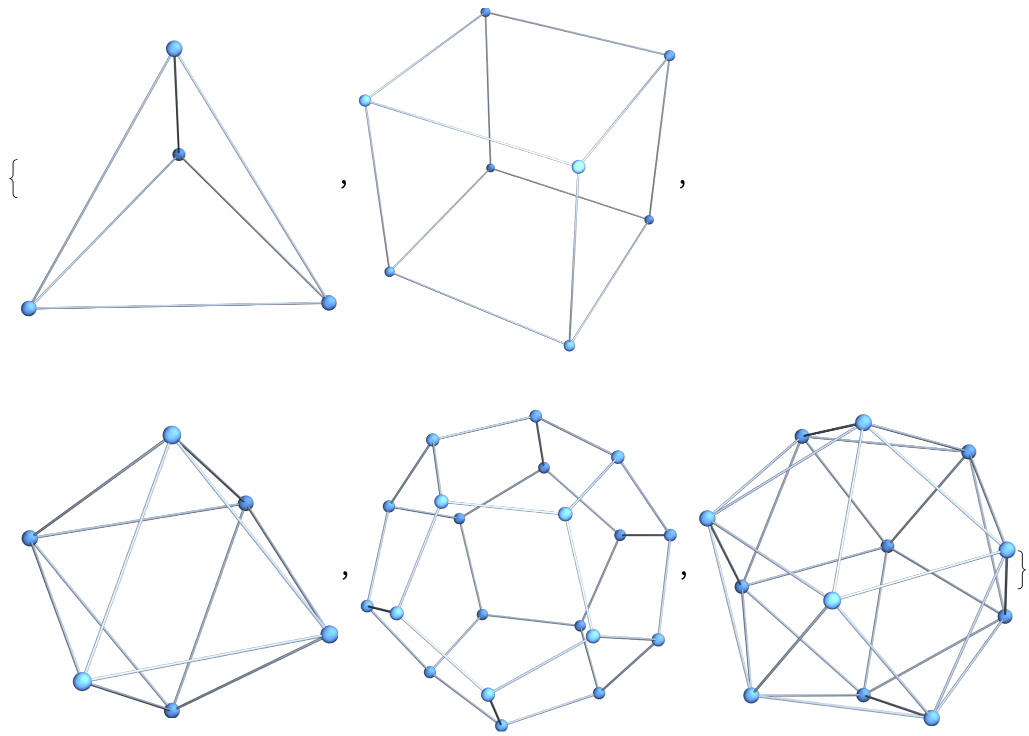
IGDistanceTransitiveQ [graph] tests if graph is distance-transitive.

`IGDistanceTransitiveQ` checks if a graph is distance transitive. In a distance transitive graph, any two ordered pairs of vertices which are the same distance apart can be mapped into each other by some automorphism.

All Platonic graphs are distance transitive.

In[861]:=
```
IGMeshGraph@PolyhedronData[#, "BoundaryMeshRegion"] & /@ PolyhedronData["Platonic"]
```

Out[861]=



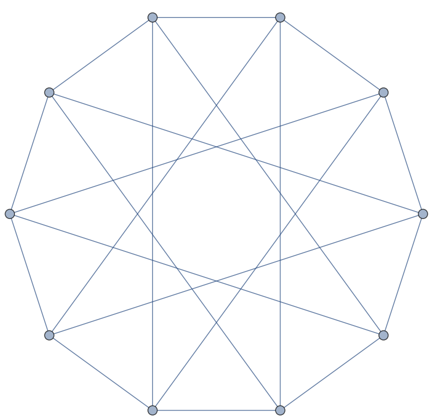
In[862]:=
```
IGDistanceTransitiveQ /@ %
```

Out[862]=
```
{True, True, True, True, True}
```

Some graphs are symmetric, but not distance transitive.

In[863]:=
```
g = GraphData[{"Circulant", {10, {1, 4}}}]
```

Out[863]=

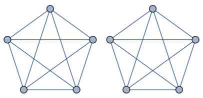


In[864]:=
```
{IGSymmetricQ[g], IGDistanceTransitiveQ[g]}
```

Out[864]=
```
{True, False}
```

`IGDistanceTransitiveQ` does not exclude non-connected graphs.

In[865]:=

```
IGDistanceTransitiveQ[
```


```
]
```

Out[865]=

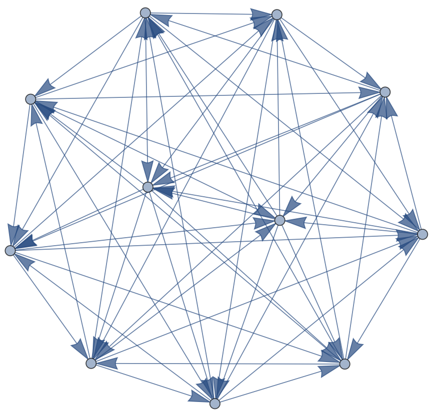
```
True
```

`IGDistanceTransitiveQ` works with directed graphs.

In[866]:=

```
g = With[{n = 11},
  RelationGraph[MemberQ[Rest@Union@Mod[Range[n]^2, n], Mod[#1 - #2, n]] &, Range[n] - 1]
 ]
```

Out[866]=



In[867]:=

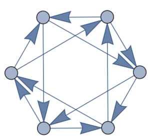
```
IGDistanceTransitiveQ[g]
```

Out[867]=

```
True
```

The following directed graph is vertex transitive, but not distance transitive.

In[868]:=

```
IGDistanceTransitiveQ[
```


```
]
```

Out[868]=

```
False
```

## Homeomorphism

In[869]:=

```
? IGHomeomorphicQ
```

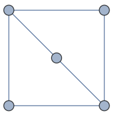
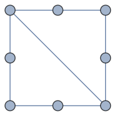
IGHomeomorphicQ[graph1, graph2] tests if graph1 and graph2 are homeomorphic. Edge directions are ignored.

`IGHomeomorphicQ` tests if two graphs are homeomorphic, i.e. whether they have the same topological structure. Two graphs $G_1$ and $G_2$ are homeomorphic if there is an isomorphism from a subdivision of $G_1$ to a subdivision of $G_2$.

`IGHomeomorphicQ[g1, g2]` is effectively implemented as `IGIsomorphicQ[IGSmoothen[g1], IGSmoothen[g2]]`.

The following graphs are homeomorphic.

In[870]:=

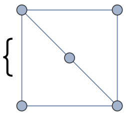
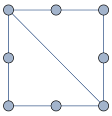
IGHomeomorphicQ[  ,  ]

Out[870]=

```
True
```

They smoothen to the same graph.

In[871]:=

IGSmoothen /@ {  ,  }

Out[871]=

{  ,  }

Any two cycle graphs are homeomorphic.

In[872]:=

```
IGHomeomorphicQ[CycleGraph[5], CycleGraph[9]]
```

Out[872]=

```
True
```

A cycle and a path graph are not homeomorphic.

In[873]:=

```
IGHomeomorphicQ[CycleGraph[5], PathGraph@Range[5]]
```

Out[873]=

```
False
```

A triangular and a square lattice on the same number of vertices are, in general, topologically different.

In[874]:=

```
IGHomeomorphicQ[IGSquareLattice[{3, 3}], IGTriangularLattice[{3, 3}]]
```

Out[874]=

```
False
```

When testing empirical graphs for equivalence, it is often useful to remove tree-like components. For example, the face-face and the face-edge adjacency graphs of a geometric mesh are equivalent, save for the tree-like components.

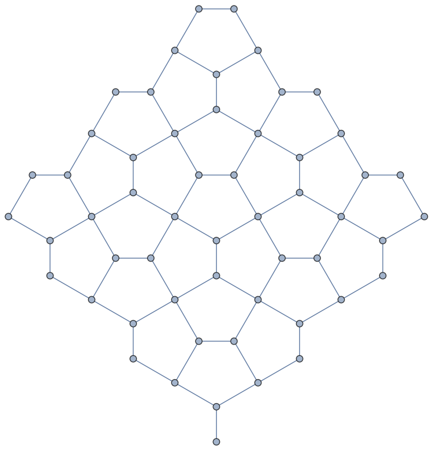
In[875]:=

```
mesh = IGLatticeMesh["SnubSquare", {3, 3}];
```

In[876]:=

```
ffg = IGMeshCellAdjacencyGraph[mesh, 2, VertexCoordinates → Automatic]
```

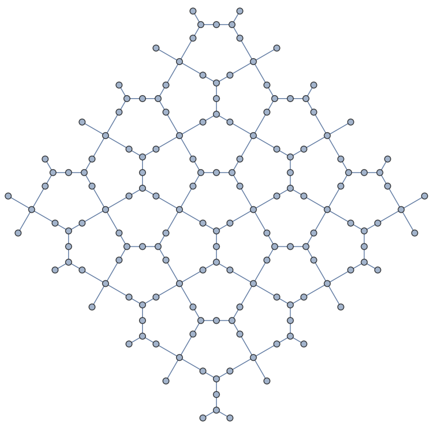
Out[876]=



In[877]:=

```
feg = IGMeshCellAdjacencyGraph[mesh, 1, 2, VertexCoordinates → Automatic]
```

Out[877]=



In[878]:=

```
IGHomeomorphicQ[feg, ffg]
```

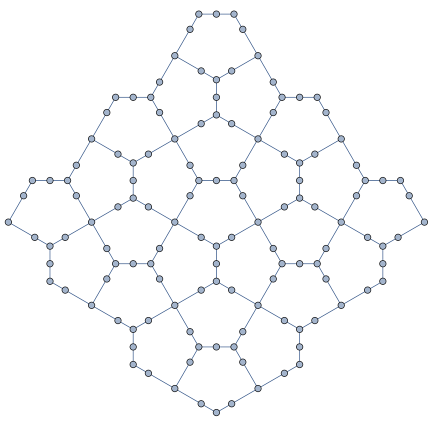
Out[878]=

```
False
```

In[879]:=

```
feg = VertexDelete[feg, IGTreelikeComponents[feg]]
```

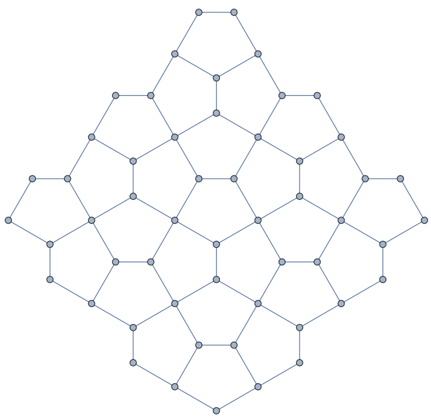
Out[879]=



In[880]:=

```
ffg = VertexDelete[ffg, IGTreelikeComponents[ffg]]
```

Out[880]=



In[881]:=

```
IGHomeomorphicQ[feg, ffg]
```

Out[881]=

```
True
```

## Other functions

### IGSelfComplementaryQ

In[882]:=

```
? IGSelfComplementaryQ
```

IGSelfComplementaryQ[graph] tests if graph is self-complementary.

A graph is called self-complementary if it is isomorphic with its complement.

The 4-vertex path graph is self-complementary.

In[883]:=

```
IGSelfComplementaryQ[PathGraph@Range[4]]
```

Out[883]=

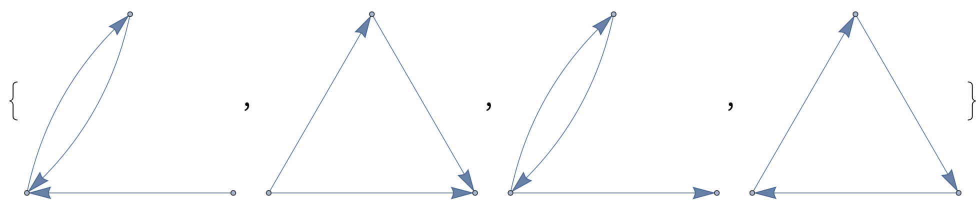
```
True
```

Find all 3-vertex self-complementary directed graphs.

In[884]:=

```
Select[IGData[{"AllDirectedGraphs", 3}], IGSelfComplementaryQ]
```

Out[884]=



## IGColoredSimpleGraph

In[885]:=

```
? IGColoredSimpleGraph
```

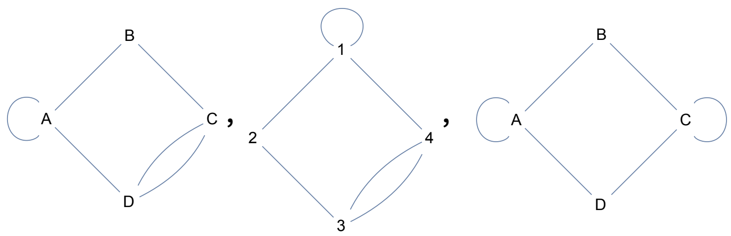
IGColoredSimpleGraph[graph] encodes a non–simple graph as an edge– and vertex–colored simple graph, returned as {simpleGraph, "VertexColors" –> vcol, "EdgeColors" –> ecol} where vertex colors represent self–loop multiplicities and edge colors represent edge multiplicities. The output is suitable for use by isomorphism functions.

`IGColoredSimpleGraph` is a helper function that encodes a non-simple graph (i.e a graph with self-loops or multi-edges) into an edge- and vertex-colored simple graph. The coloured simple graph can be used directly as an input to coloured isomorphism checking functions such as `IGVF2IsomorphicQ`.

The vertex colours are computed as the multiplicity of self-loops at each vertex. The edge colours are computed as the multiplicities or non-loop edges.

The following graphs are not simple and cannot be used with `IGVF2IsomorphicQ` directly.

In[886]:=

`{g1, g2, g3} = {`  `};`



In[887]:=

```
IGVF2IsomorphicQ[g1, g2]
```

IGraphM: VF2 does not support non–simple graphs. Consider using IGIsomorphicQ or IGColoredSimpleGraph.

Out[887]=

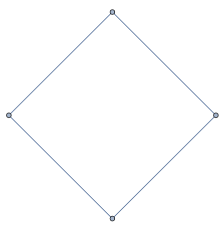
```
$Failed
```

`IGColoredSimpleGraph` can encode them as coloured graphs. Its output can be supplied directly to `IGVF2IsomorphicQ`.

In[888]:=

```
IGColoredSimpleGraph[g1]
```

Out[888]=

`{`  `, VertexColors → {1, 0, 0, 0}, EdgeColors → {1, 1, 1, 2}}`

Now can can determine that `g1` is isomorphic to `g2`, but not to `g3`.

In[889]:=

```
IGVF2IsomorphicQ[IGColoredSimpleGraph[g1], IGColoredSimpleGraph[g2]]
```

Out[889]=

```
True
```

In[890]:=

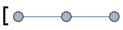
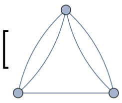
```
IGVF2IsomorphicQ[IGColoredSimpleGraph[g1], IGColoredSimpleGraph[g3]]
```

Out[890]=

```
False
```

When searching for subgraphs in multigraphs with this method, be aware that a match occurs only if the edge multiplicities are the same. This sort of matching is useful e.g. in substructure search chemistry, where a double bond must only match another double bond, but not a single one.

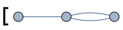
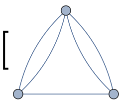
In[891]:=

`IGVF2SubisomorphicQ[IGColoredSimpleGraph[`  `], IGColoredSimpleGraph[`  `]]`

Out[891]=

`False`

In[892]:=

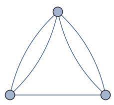
`IGVF2SubisomorphicQ[IGColoredSimpleGraph[`  `], IGColoredSimpleGraph[`  `]]`

Out[892]=

`True`

Use `IGSubisomorphicQ` to match any subgraph.

In[893]:=

`IGSubisomorphicQ[`  `,`  `]`

Out[893]=

`True`

# Maximum flow and minimum cut

## Maximum flow

### IGMaximumFlowValue

In[894]:=

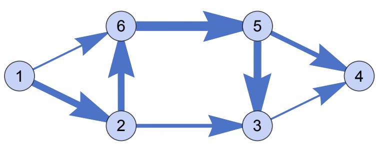
```
? IGMaximumFlowValue
```

IGMaximumFlowValue[graph, s, t] gives the value of the maximum flow from s to t.

`IGMaximumFlowValue` is equivalent to `IGMinimumCutValue` except that it uses the `EdgeCapacity` property instead of `EdgeWeight`.

Edge capacities are taken from the `EdgeCapacity` property.

In[895]:=

`g =` 

`;`

```
IGEdgeProp[EdgeCapacity][g]
```

Out[896]=

```
{3.5, 2, 1, 2.5, 5, 1, 3.5, 4}
```

In[897]:=

```
IGMaximumFlowValue[g, 1, 4]
```

Out[897]=

```
3.5
```

## IGMaximumFlowMatrix

In[898]:=

```
? IGMaximumFlowMatrix
```

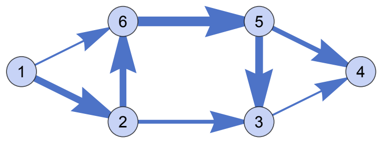
IGMaximumFlowMatrix[graph, s, t] gives the flow matrix of a maximum flow from s to t.

Element $F_{ij}$ of the flow matrix is the flow through the edge connecting the $i$th node to the $j$th one. In an undirected graph, $F_{ij} = -F_{ij}$.

Edge capacities are taken from the `EdgeCapacity` property.

Let us take a directed graph with edge capacities set ...

In[899]:=

g =  ;



In[900]:=

```
IGEdgeProp[EdgeCapacity][g]
```

Out[900]=

```
{3.5, 2, 1, 2.5, 5, 1, 3.5, 4}
```

... and compute the maximum flow between two of its vertices.

In[901]:=

```
flowMat = IGMaximumFlowMatrix[g, 1, 4]
```

Out[901]=

```
SparseArray[ Specified elements: 6
             Dimensions: {6, 6} ]
```

The result is returned as a sparse matrix containing the flows through each edge.

In[902]:=

```
MatrixForm[flowMat]
```

Out[902]//MatrixForm=

$$\begin{pmatrix} 0. & 3.5 & 0. & 0. & 0. & 0. \\ 0. & 0. & 1. & 0. & 0. & 2.5 \\ 0. & 0. & 0. & 1. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 2.5 & 0. & 0. \\ 0. & 0. & 0. & 0. & 2.5 & 0. \end{pmatrix}$$

If the input is an undirected graph, the flow matrix contains entries of opposing sign for the two directions along each edge.

In[903]:=

```
IGMaximumFlowMatrix[UndirectedGraph[g], 1, 4] // MatrixForm
```

Out[903]//MatrixForm=

$$\begin{pmatrix} 0. & 2.5 & 0. & 0. & 0. & 1. \\ -2.5 & 0. & 2. & 0. & 0. & 0.5 \\ 0. & -2. & 0. & 1. & 1. & 0. \\ 0. & 0. & -1. & 0. & -2.5 & 0. \\ 0. & 0. & -1. & 2.5 & 0. & -1.5 \\ -1. & -0.5 & 0. & 0. & 1.5 & 0. \end{pmatrix}$$

## Minimum edge cuts

### IGMinimumCut

In[904]:=

```
? IGMinimumCut
```

IGMinimumCut[graph] gives a minimum edge cut in a weighted graph.
IGMinimumCut[graph, s, t] gives a minimum s–t edge cut in a weighted graph.

`IGMinimumCut` finds a single minimum edge cut in a weighted graph. To find all minimum cuts between two given vertices, use `IGFindMinimumCuts`.

### IGMinimumCutValue

In[905]:=

```
? IGMinimumCutValue
```

IGMinimumCutValue[graph] gives the smallest sum of weights corresponding to an edge cut in graph.
IGMinimumCutValue[graph, s, t] gives the smallest sum of weights corresponding to an s–t edge cut in graph.

Unlike `IGEdgeConnectivity`, `IGMinimumCutValue` takes weights into account.

In[906]:=

```
IGMinimumCutValue[Graph[{1 ↔ 2, 2 ↔ 3}, EdgeWeight → {3.5, 5.6}]]
```

Out[906]=

```
3.5
```

The minimum cut value of the null graph and singleton graph are returned as `0` and ∞, respectively.

In[907]:=

```
IGMinimumCutValue /@ {IGEmptyGraph[0], IGEmptyGraph[1]}
```

Out[907]=

```
{0., ∞}
```

### IGGomoryHuTree

In[908]:=

```
? IGGomoryHuTree
```

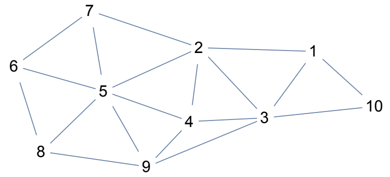
IGGomoryHuTree[graph] gives the Gomory–Hu tree of a graph.

The Gomory–Hu tree is a weighted tree that encodes the minimum cuts between all pairs of vertices of an undirected graph. The Gomory–Hu tree has the same vertices as the graph it characterizes. The minimum cut between an *s*-*t* pair of the graph has the same size as smallest edge weight on the path from *s* to *t* in the Gomory–Hu tree.

Weighted graphs are supported.

In[909]:=

```
g = 
```



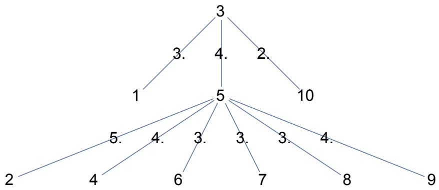


```
;
```

In[910]:=

```
t = IGGomoryHuTree[g,
  EdgeLabels → "EdgeWeight", VertexShapeFunction → "Name"
 ]
```

Out[910]=





The path from 1 to 9 is 1 ⟷ 2 ⟷ 5 ⟷ 9 and has the weights {3, 5, 4}. The smallest one, 3, is the minimum value of a cut separating 1 from 9.

In[911]:=

```
{IGMinimumCutValue[g, 1, 9], IGMinimumCutValue[t, 1, 9]}
```

Out[911]=

```
{3., 3.}
```

## Cohesive blocks

In[912]:=

```
? IGCohesiveBlocks
```

IGCohesiveBlocks[graph] gives the cohesive block structure of a simple undirected graph.

The following examples are based on the ones in the R/igraph documentation.

This is the network from the Moody-White paper:

- J. Moody and D. R. White. Structural cohesion and embeddedness: A hierarchical concept of social groups. American Sociological Review, 68(1):103–127, Feb 2003.

In[913]:=

```
mw = Graph[{"1" ⟷ "2", "1" ⟷ "3", "1" ⟷ "4", "1" ⟷ "5", "1" ⟷ "6", "2" ⟷ "3", "2" ⟷ "4", "2" ⟷ "5",
    "2" ⟷ "7", "3" ⟷ "4", "3" ⟷ "6", "3" ⟷ "7", "4" ⟷ "5", "4" ⟷ "6", "4" ⟷ "7", "5" ⟷ "6",
    "5" ⟷ "7", "5" ⟷ "21", "6" ⟷ "7", "7" ⟷ "8", "7" ⟷ "11", "7" ⟷ "14", "7" ⟷ "19",
    "8" ⟷ "9", "8" ⟷ "11", "8" ⟷ "14", "9" ⟷ "10", "10" ⟷ "12", "10" ⟷ "13", "11" ⟷ "12",
    "11" ⟷ "14", "12" ⟷ "16", "13" ⟷ "16", "14" ⟷ "15", "15" ⟷ "16", "17" ⟷ "18",
    "17" ⟷ "19", "17" ⟷ "20", "18" ⟷ "20", "18" ⟷ "21", "19" ⟷ "20", "19" ⟷ "22",
    "19" ⟷ "23", "20" ⟷ "21", "21" ⟷ "22", "21" ⟷ "23", "22" ⟷ "23"}, VertexLabels → "Name"];
```

In[914]:=

```
{blocks, cohesion} = IGCohesiveBlocks[mw]
```

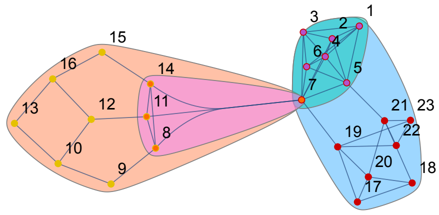
Out[914]=

```
{{{1, 2, 3, 4, 5, 6, 7, 21, 8, 11, 14, 19, 9, 10, 12, 13, 16, 15, 17, 18, 20, 22, 23},
  {1, 2, 3, 4, 5, 6, 7, 21, 19, 17, 18, 20, 22, 23}, {7, 8, 11, 14, 9, 10, 12, 13, 16, 15},
  {1, 2, 3, 4, 5, 6, 7}, {7, 8, 11, 14}}, {1, 2, 2, 5, 3}}
```

In[915]:=

```
CommunityGraphPlot[mw, Rest@blocks,
 CommunityRegionStyle → Table[Directive[Opacity[0.5], ColorData[96][i]], {i, Length[blocks] - 1}]]
```

Out[915]=




In[916]:=

```
cohesion
```

Out[916]=

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

Science camp network:

In[917]:=

```
sc = Graph[{"Pauline" ↔ "Jennie", "Pauline" ↔ "Ann", "Jennie" ↔ "Ann", "Jennie" ↔ "Michael",
    "Michael" ↔ "Ann", "Holly" ↔ "Jennie", "Jennie" ↔ "Lee", "Michael" ↔ "Lee",
    "Harry" ↔ "Bert", "Harry" ↔ "Don", "Don" ↔ "Bert", "Gery" ↔ "Russ", "Russ" ↔ "Bert",
    "Michael" ↔ "John", "Gery" ↔ "John", "Russ" ↔ "John", "Holly" ↔ "Pam", "Pam" ↔ "Carol",
    "Holly" ↔ "Carol", "Holly" ↔ "Bill", "Bill" ↔ "Pauline", "Bill" ↔ "Michael",
    "Bill" ↔ "Lee", "Harry" ↔ "Steve", "Steve" ↔ "Don", "Steve" ↔ "Bert", "Gery" ↔ "Steve",
    "Russ" ↔ "Steve", "Pam" ↔ "Brazey", "Brazey" ↔ "Carol", "Pam" ↔ "Pat", "Brazey" ↔ "Pat",
    "Carol" ↔ "Pat", "Holly" ↔ "Pat", "Gery" ↔ "Pat"}, VertexLabels → "Name"];
```

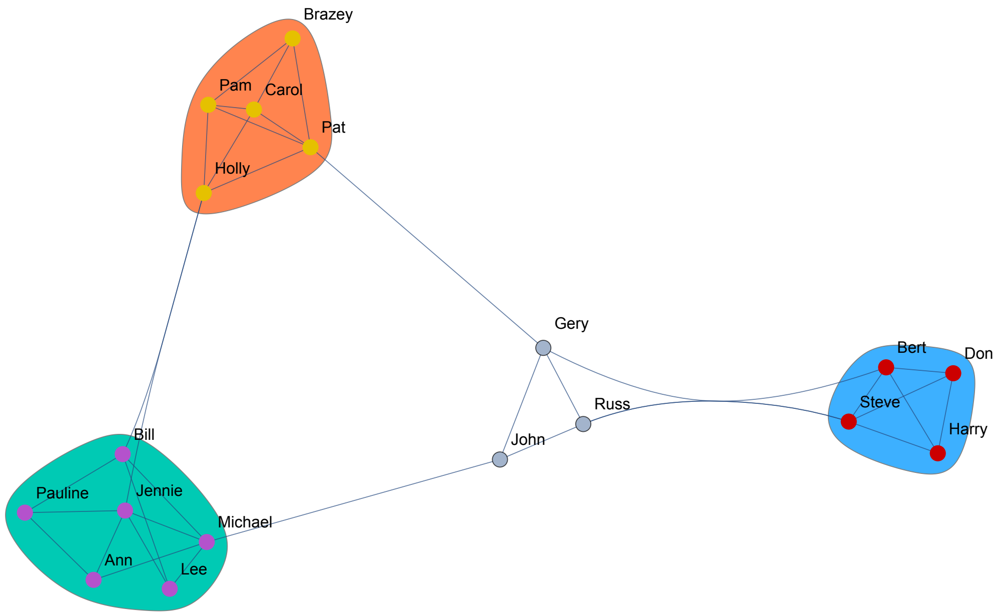
In[918]:=

```
{blocks, cohesion} = IGCohesiveBlocks[sc]
```

Out[918]=

```
{{{Pauline, Jennie, Ann, Michael, Holly, Lee, Harry, Bert, Don, Gery,
   Russ, John, Pam, Carol, Bill, Steve, Brazey, Pat}, {Harry, Bert, Don, Steve},
  {Holly, Pam, Carol, Brazey, Pat}, {Pauline, Jennie, Ann, Michael, Lee, Bill}}, {2, 3, 3, 3}}
```

In[919]:=
```
CommunityGraphPlot[sc, Rest@blocks, CommunityRegionStyle → ColorData[96], ImageSize → Large]
```
Out[919]=




# Cliques and independent vertex sets

In[920]:=
```
? IG*Clique*
```

**IGraphM`**

| | | |
|---|---|---|
| IGCliqueCover | IGLargestCliques | IGMaximalWeightedCliques |
| IGCliqueCoverNumber | IGLargestWeightedCliques | IGWeightedCliqueNumber |
| IGCliqueNumber | IGMaximalCliques | IGWeightedCliques |
| IGCliques | IGMaximalCliquesCount | |
| IGCliqueSizeCounts | IGMaximalCliqueSizeCounts | |

A clique is a fully connected subgraph. An independent vertex set is a subset of a graph's vertices with no connections between them.

## Counting cliques

*Mathematica*'s `FindClique` function only finds maximal cliques. IGraph/M provides functions for finding or counting all cliques, i.e. complete subgraphs, of a graph.

In[921]:=
```
g = ExampleData[{"NetworkGraph", "CoauthorshipsInNetworkScience"}];
```

In[922]:=
```
{VertexCount[g], EdgeCount[g]}
```

Out[922]=
```
{1589, 2742}
```

Simply counting cliques is much more memory efficient (and faster) than returning all of them.

In[923]:=

```
IGCliqueSizeCounts[g]
```

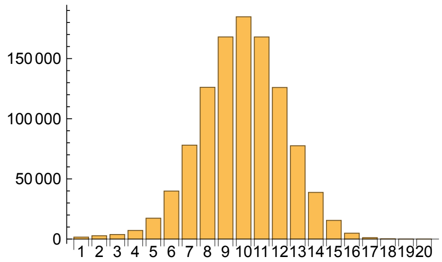
Out[923]=

```
{1589, 2742, 3764, 7159, 17 314, 39 906, 78 055, 126 140, 167 993,
 184 759, 167 960, 125 970, 77 520, 38 760, 15 504, 4845, 1140, 190, 20, 1}
```

In[924]:=

```
BarChart[%, ChartLabels → Range@Length[%]]
```

Out[924]=




In[925]:=

```
IGMaximalCliqueSizeCounts[g]
```

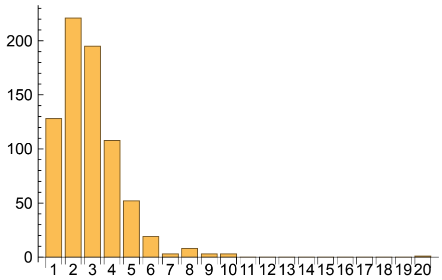
Out[925]=

```
{128, 221, 195, 108, 52, 19, 3, 8, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
```

In[926]:=

```
BarChart[%, ChartLabels → Range@Length[%]]
```

Out[926]=




In[927]:=

```
IGLargestCliques[g]
```

Out[927]=

```
{{V. Narayan, L. Giot, J. Rothberg, S. Fields, M. Johnston, M. Yang, G. Vijayadamodar,
  T. Kalbfleisch, D. Conover, B. Godwin, Y. Li, A. Qureshiemili, P. Pochart,
  M. Srinivasan, D. Lockshon, J. Knight, R. Judson, T. Mansfield, G. Cagney, P. Uetz}}
```

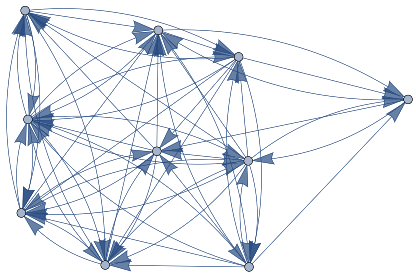
## Cliques in directed graphs

The clique finder in IGraph/M ignores edge directions.

In[928]:=

```
g = RandomGraph[{10, 60}, DirectedEdges → True]
```

Out[928]=



In[929]:=

```
IGMaximalCliques[g]
```

IGraphM: src/cliques/maximal_cliques.c:269 – Edge directions are ignored for maximal clique calculation.

Out[929]=

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

To find cliques in directed graphs, convert them to undirected and keep mutual (bidirectional) edges only.

In[930]:=

```
IGMaximalCliques@IGUndirectedGraph[g, "Mutual"]
```

Out[930]=

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

## Clique cover

In[931]:=

```
? IGCliqueCover
```

IGCliqueCover[graph] gives a minimum clique cover of graph, i.e. a partitioning of its vertices into a smallest number of cliques.

In[932]:=

```
? IGCliqueCoverNumber
```

IGCliqueCoverNumber[graphs] gives the clique vertex cover number of graph.

A clique cover of a graph is a partitioning of its vertices such that each partition forms a clique. `IGCliqueCover` finds a minimum clique cover, i.e. a partitioning into a smallest number of cliques.

The clique cover number of a graph is the smallest number of cliques that can be used to cover its vertices.

Available `Method` option values are:

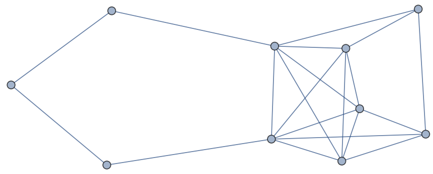
- `"Minimum"` finds a minimum clique cover.
- `"Heuristic"` is much faster, but the result is not typically a minimum cover.

Compute a minimum clique cover of a random graph.

In[933]:=

```
g = RandomGraph[{10, 20}]
```

Out[933]=



In[934]:=

```
IGCliqueCover[g]
```

Out[934]=

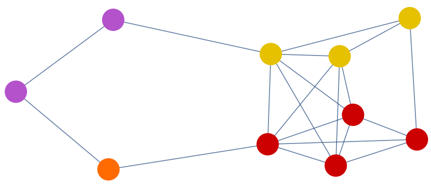
```
{{1, 2, 6, 7}, {3, 5, 9}, {4, 10}, {8}}
```

Visualize the clique cover.

In[935]:=

```
HighlightGraph[g, IGCliqueCover[g], VertexSize → Large]
```

Out[935]=



Find the clique cover number without returning a cover.

In[936]:=

```
IGCliqueCoverNumber[g]
```

Out[936]=

```
4
```

The clique cover problem is equivalent to the colouring of the complement graph. `IGCliqueCover` is effectively implemented as

In[937]:=

```
IGMembershipToPartitions[g]@IGMinimumVertexColoring@GraphComplement[g]
```

Out[937]=

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

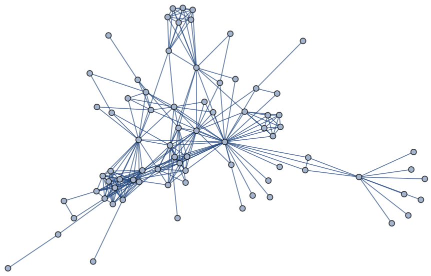
For difficult problems, it may be useful to use `IGMinimumVertexColoring` or `IGVertexColoring` directly instead of `IGCliqueCover`, and tune their options to achieve better performance. See the `"ForcedColoring"` option of `IGMinimumVertexColoring` on how to do this.

## Reconstruct bipartite graph of co-occurrence network

In[938]:=

```
g = ExampleData[{"NetworkGraph", "LesMiserables"}]
```

Out[938]=



In[939]:=

```
ExampleData[{"NetworkGraph", "LesMiserables"}, "LongDescription"]
```

Out[939]=

```
Coappearance network of characters in the novel
  Les Miserables. EdgeWeight describes the number of coappearance.
```

The maximal cliques of the graph can approximate the scenes in which characters appear together.

In[940]:=

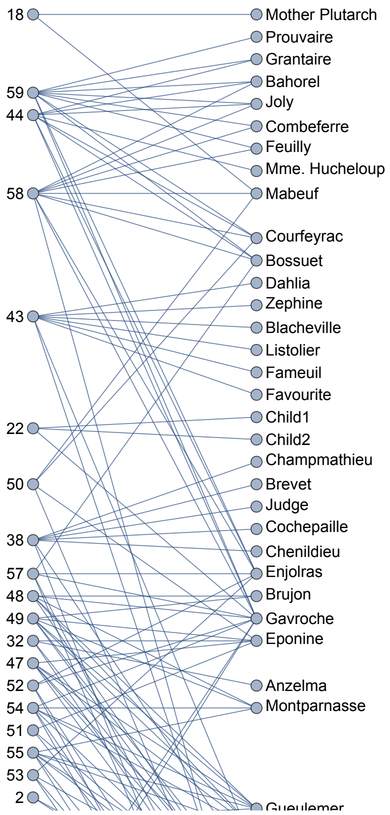
```
cliques = IGMaximalCliques[g];
```

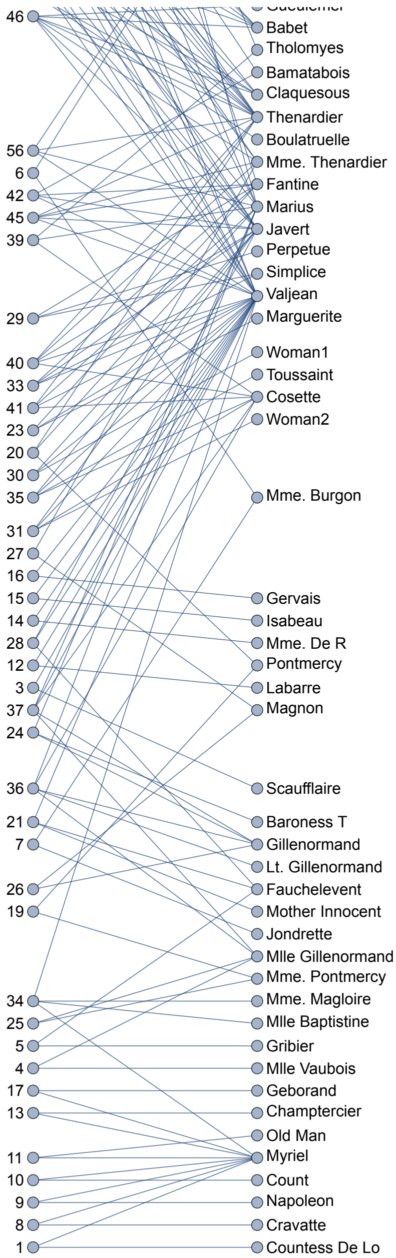
We can construct a bipartite graph of connections between potential scenes and characters

In[941]:=

```
IGLayoutBipartite[
  Graph@Catenate[Thread /@ Thread[Range@Length[cliques] ↔ cliques]],
  VertexSize → 0.5, ImageSize → 220
 ] // IGVertexMap[Placed[#, If[IntegerQ[#], Before, After]] &, VertexLabels → VertexList]
```

Out[941]=






## Graphlet decomposition

***Note:*** *The term “graphlet” is used for multiple unrelated concepts in the literature. This section deals with decomposing weighted graphs into cliques. If you are looking to count induced subgraphs, see the* `IGMotifs` *function.*

In[942]:=

```
? IGGraphlets
```

IGGraphlets[graph] decomposes a weighted graph into a sum of cliques.

In[943]:=

```
? IGGraphletBasis
```

IGGraphletBasis[graph] computes a candidate clique basis.

In[944]:=

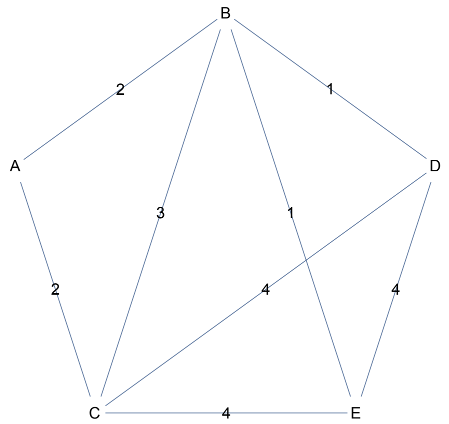
```
? IGGraphletProject
```

IGGraphletProject[graph, cliques] projects a weighted graph onto the given clique basis.

In[945]:=

```
g = IGShorthand["A,B,D,E,C, A-B-C-A, C-E-D-B, D-C, E-B",
  EdgeWeight → {2, 3, 2, 4, 4, 1, 4, 1},
  EdgeLabels → "EdgeWeight", VertexLabels → None,
  VertexShapeFunction → "Name", PerformanceGoal → "Quality",
  GraphLayout → "CircularEmbedding"
 ]
```

Out[945]=




In[946]:=

```
basis = IGGraphletBasis[g]
```

Out[946]=

```
<|{A, B, C} → 2., {B, D, E, C} → 1., {B, C} → 3., {D, E, C} → 4.|>
```

In[947]:=

```
IGGraphletProject[g, Keys[basis]]
```

Out[947]=

<|{A, B, C} → 0.925543, {B, D, E, C} → 0.861478, {B, C} → $2.90881 \times 10^{-253}$, {D, E, C} → 1.13842|>

In[948]:=

```
IGGraphlets[g]
```

Out[948]=

<|{D, E, C} → 1.13842, {A, B, C} → 0.925543, {B, D, E, C} → 0.861478, {B, C} → $2.90881 \times 10^{-253}$|>

## References


- Hossein Azari Soufiani and Edoardo M Airoldi, Graphlet decomposition of a weighted network, https://arxiv.org/abs/1203.2821

# Layout algorithms

The following functions are available:

In[949]:=

```
? IGLayout*
```

IGraphM`

| | | |
|---|---|---|
| IGLayoutBipartite | IGLayoutFruchtermanReingold3D | IGLayoutRandom |
| IGLayoutCircle | IGLayoutGEM | IGLayoutReingoldTilford |
| IGLayoutDavidsonHarel | IGLayoutGraphOpt | IGLayoutReingoldTilfordCircular |
| IGLayoutDrL | IGLayoutKamadaKawai | IGLayoutSphere |
| IGLayoutDrL3D | IGLayoutKamadaKawai3D | IGLayoutTutte |
| IGLayoutFruchtermanReingold | IGLayoutPlanar | |

If you are looking for the Sugiyama layout from igraph, try the built-in `GraphLayout → "LayeredDigraphEmbedding"`, or `LayeredGraphPlot`. These are also based on the Sugiyama algorithm.

## Common options and examples

Layout functions also take any standard `Graph` option.

Many layout algorithms take the following options:

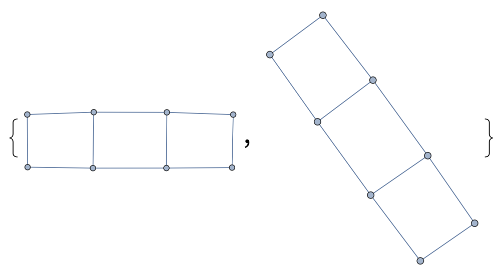
`"MaxIterations"` controls either the *maximum* number of iterations performed by the algorithm or the *exact* number of iterations, depending on the specific algorithm and settings. The option name is the same for all functions to make it easier to interchange them when visualizing dynamic graphs.

`"Align" → True` aligns the output horizontally. Examples:

In[950]:=

```
{IGLayoutFruchtermanReingold[IGSquareLattice[{2, 4}](*, "Align" → True is the default *)],
 IGLayoutFruchtermanReingold[IGSquareLattice[{2, 4}], "Align" → False]}
```

Out[950]=

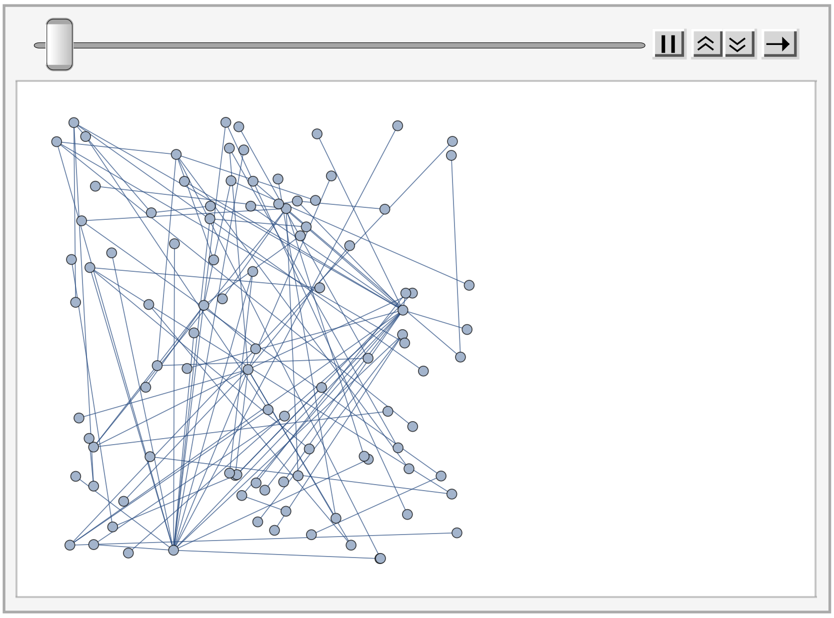


`"Continue" → True` allows using existing vertex coordinates as starting points for algorithms that update vertex positions incrementally. We can use this to visualize how the layout algorithms work …

In[951]:=

```
g = IGLayoutRandom@RandomGraph[BarabasiAlbertGraphDistribution[100, 1]];
ListAnimate@
 NestList[IGLayoutGraphOpt[#, "Continue" → True, "MaxIterations" → 80, "Align" → False] &, g, 40]
```

Out[952]=

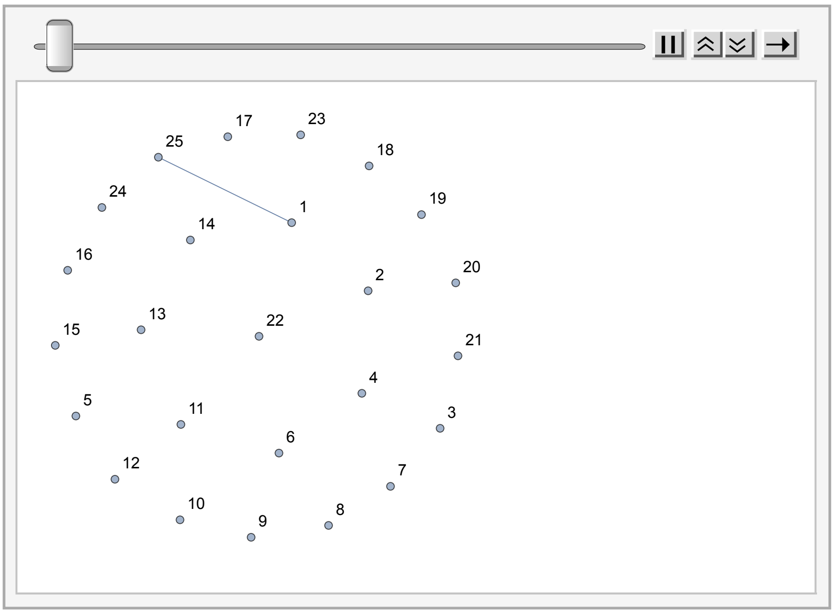


… or to visualize dynamic graph processes such as adding edges to the graph one by one:

In[953]:=

```
g = IGLayoutKamadaKawai@Graph[Range[25], {1 ⟷ 25}, VertexLabels → "Name"];
ListAnimate@NestList[
  IGLayoutKamadaKawai[EdgeAdd[#, UndirectedEdge @@ RandomSample[VertexList[#], 2]],
    "MaxIterations" → 15, "Continue" → True, "Align" → False] &,
  g,
  30
 ]
```

Out[954]=

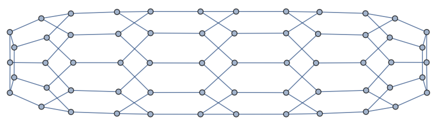



Visualize a planar graph without edge crossings using the Davidson–Harel simulated annealing method, and taking starting coordinates from `GraphLayout → "PlanarEmbedding"`.

In[955]:=

```
g = Graph@GraphData[{"Fullerene", {60, 1}}, "EdgeList"]
```

Out[955]=

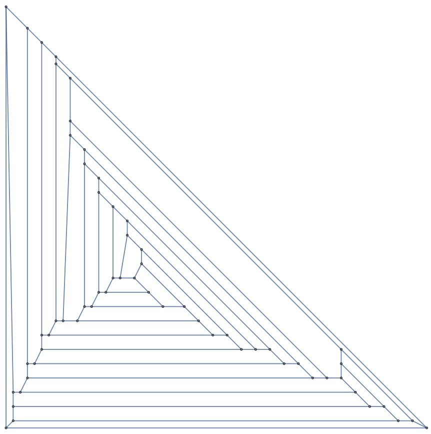


This layout avoids crossings, but it is not pleasing:

In[956]:=

```
Graph[g, GraphLayout → "PlanarEmbedding"]
```

Out[956]=

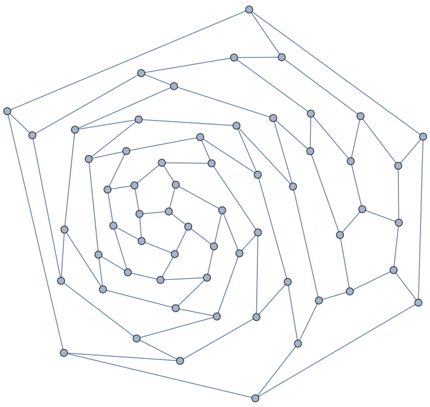


We can post process it while avoiding the introduction of any edge crossings:

In[957]:=

```
IGLayoutDavidsonHarel[
 IGVertexMap[# &, VertexCoordinates → (Rescale@GraphEmbedding[#, "PlanarEmbedding"] &), g],
 "Continue" → True, "EdgeCrossingWeight" → 1000
]
```

Out[957]=



## Weighted graphs

Several of the graph layout algorithms in igraph can take edge weights into accounts. How the weights are used during layout differs between them.

- IGLayoutFruchtermanReingold multiplies the attraction between vertices by the weights. Thus higher weights result in shorter edges.
- IGLayoutKamadaKawai produces longer edges for higher weights

## Constraining vertex positions

Graph layout functions which have a "Constraints" option allow fixing the position of some vertices, or constraining them into a box. This is an experimental feature that may change in the future.

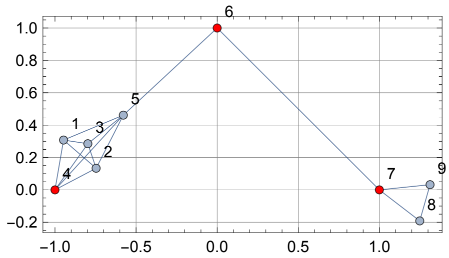
The value of the "Constraints" option must be an association from vertex names to vertex coordinates, or to bounding boxes.

Fix the positions of three vertices and highlight them in red:

In[958]:=

```
IGLayoutFruchtermanReingold[
 IGShorthand["1:2:3:4 - 1:2:3:4:5, 5-6-7, 7:8:9 - 7:8:9"],
 "Constraints" → <|4 → {-1, 0}, 7 → {1, 0}, 6 → {0, 1}|>,
 VertexStyle → {4 → Red, 7 → Red, 6 → Red},
 Frame → True, FrameTicks → True, GridLines → Automatic
]
```

Out[958]=




## Drawing trees

`IGLayoutReingoldTilford[]` and `IGLayoutReingoldTilfordCircular[]` are designed for laying out trees or forests. The following options are available:

- `"RootVertices"` allows specifying the root node(s). It must be a list, even if there is a single root node. Multiple root nodes are meant to be used with forests. The roots should be selected so that all vertices of the graph are reachable from them. `"RootVertices" → Automatic` chooses roots automatically, preferring low eccentricity vertices in small graphs (fewer than 500 vertices) and high degree vertices in large graphs.
- `DirectedEdges → False` ignores edge directions. By default, directed graphs are laid out so that edges are pointing away from the root.
- `"Rotation"` controls the orientation of the layout. It must be given in radians.

The following options are unique to `IGLayoutReingoldTilford[]`:

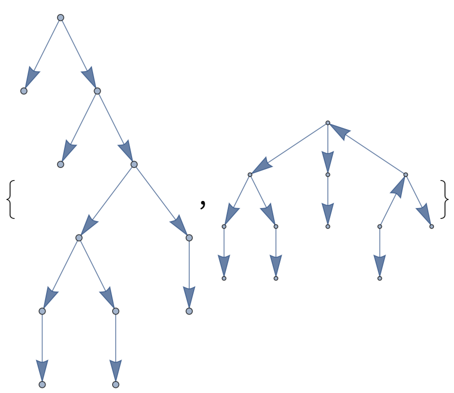
- `"LeafDistance"` sets the spacing between tree leaves. The default is 1.
- `"LayerHeight"` sets the spacing between layers of the drawing. The default is 1.

The same tree laid out in directed and undirected modes:

In[959]:=

```
t = IGTreeGame[12, DirectedEdges → True];
{IGLayoutReingoldTilford[t],
 IGLayoutReingoldTilford[t, DirectedEdges → False]}
```

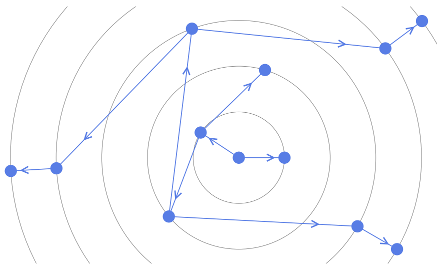
Out[960]=



Lay out the tree radially, with successive layers places on circular shells:

In[961]:=
```
IGLayoutReingoldTilfordCircular[t,
 GraphStyle → "Minimal",
 Prolog → {Thin, Gray, Table[Circle[{0, 0}, r], {r, IGDiameter[t, "ByComponents" → True]}]}]
```

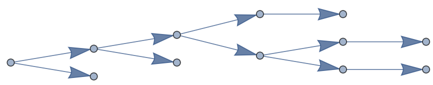
Out[961]=



Use a left-to-right layout, with tightly spaced leaves:

In[962]:=
```
IGLayoutReingoldTilford[t, "Rotation" → Pi / 2, "LeafDistance" → 1 / 3]
```

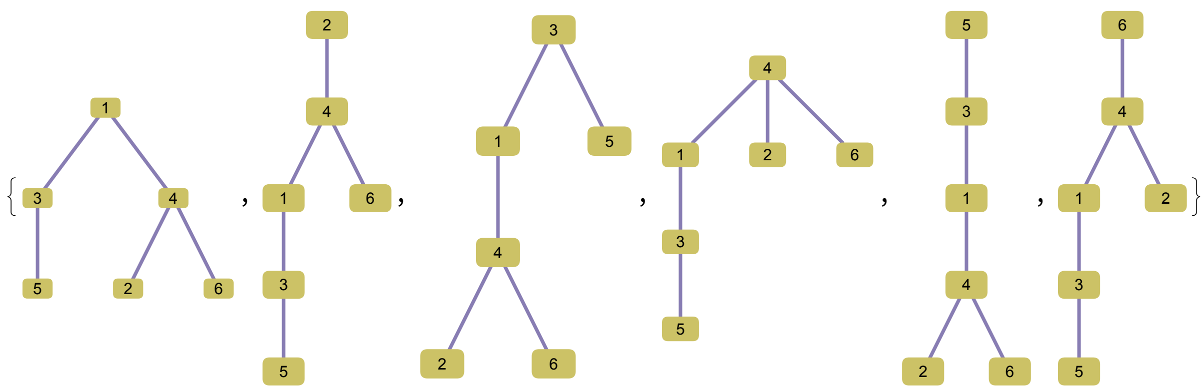
Out[962]=



In an undirected tree, any vertex may be chosen as the root:

In[963]:=
```
t = IGTreeGame[6];
Table[
 IGLayoutReingoldTilford[t, "RootVertices" → {r}, GraphStyle → "DiagramGold"],
 {r, VertexList[t]}
]
```

Out[964]=




## Drawing bipartite graphs

In[965]:=
```
? IGLayoutBipartite
```

IGLayoutBipartite[graph, options] lays out a bipartite graph, minimizing the number of edge crossings. Partitions can be specified manually using the "BipartitePartitions" option.

`IGLayoutBipartite` draws a bipartite graph, attempting to minimize the number of edge crossing using the Sugiyama algorithm.

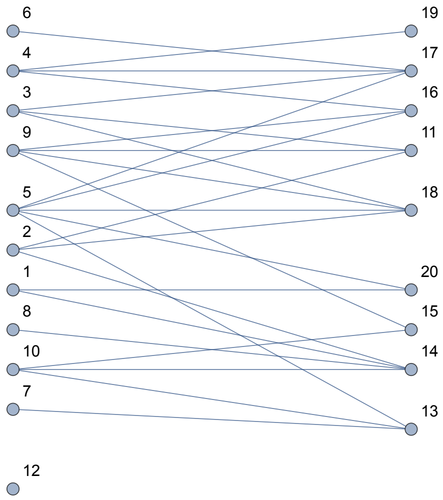
The available options are:

- "Orientation" can be Horizontal or Vertical
- "PartitionGap" controls the size of the gap between the two partitions
- "VertexGap" controls the minimum size of the gap between vertices in a partition
- MaxIterations controls the maximum number of iterations performed during edge crossing minimization.
- "BipartitePartitions" can be used to explicitly specify the partitioning of the graph.

In[966]:=

```
IGLayoutBipartite[IGBipartiteGameGNP[10, 10, 0.2], VertexLabels → "Name"]
```

Out[966]=

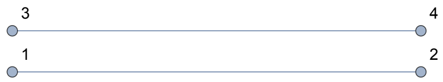



By default, a partitioning is computed automatically.

In[967]:=

```
g = Graph[{1 ⟷ 2, 3 ⟷ 4}, VertexLabels → "Name"];
IGLayoutBipartite[g]
```

Out[968]=




The partitioning can also be specified explicitly.

In[969]:=

```
IGLayoutBipartite[g, "BipartitePartitions" → {{2, 3}, {4, 1}}]
```

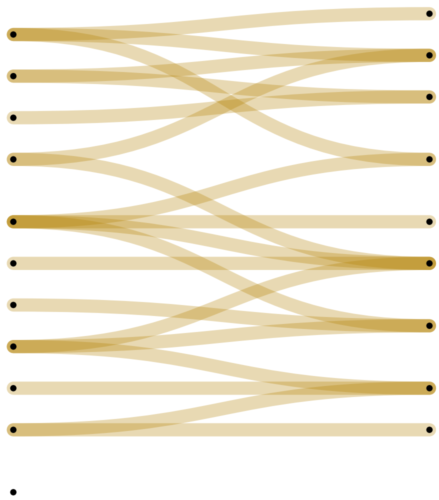
Out[969]=

Draw a bipartite layout with curved edges.

In[970]:=

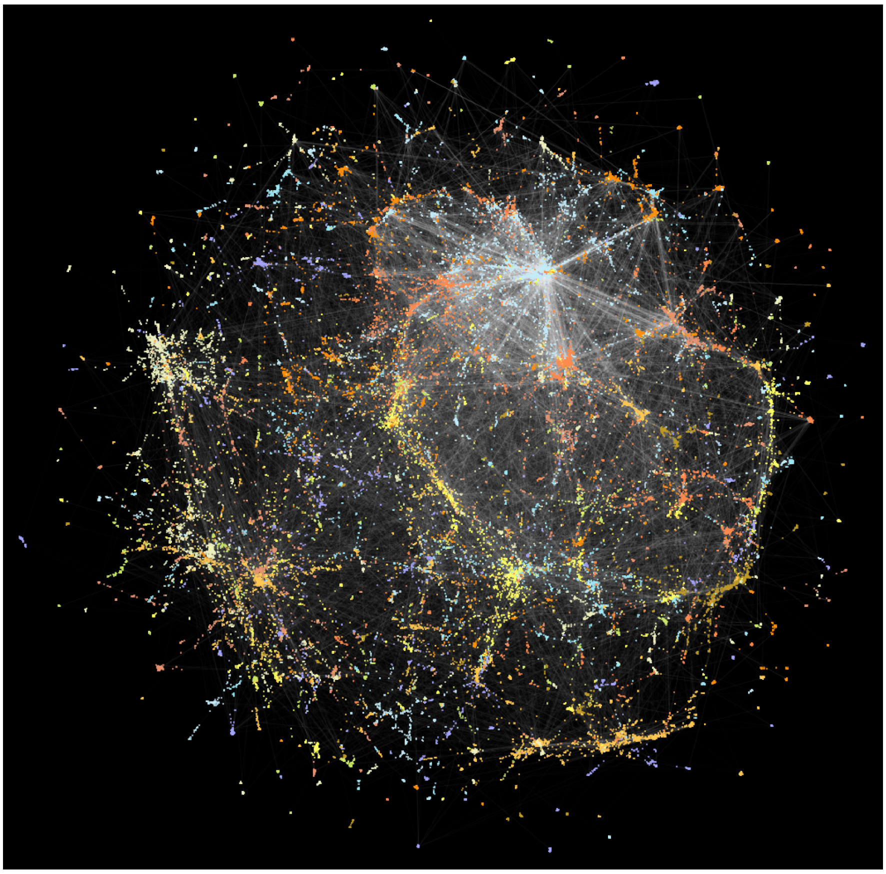
```
snake[{x1_, y1_}, {x2_, y2_}] := BezierCurve[{{x1, y1}, {(x1 + x2)/2, y1}, {(x1 + x2)/2, y2}, {x2, y2}}]
IGLayoutBipartite@IGBipartiteGameGNM[10, 10, 20,
  EdgeShapeFunction → ({CapForm["Round"], snake[First[#1], Last[#1]]} &),
  GraphStyle → "ThickEdge", VertexStyle → Black]
```

Out[971]=



## Drawing large graphs

IGLayoutDrL is designed specifically for visualizing large graphs with high clustering. The following image is created using DrL and shows a 36 000 node network of collaborations between condensed matter scientists.



The image was generated using the following code:

```
lg = ExampleData[{"NetworkGraph", "CondensedMatterCollaborations2005"}];
lg = IndexGraph@Subgraph[lg, First@ConnectedComponents[lg]];
c = IGCommunitiesMultilevel[lg]
pts = GraphEmbedding@IGLayoutDrL[lg]; (* this takes a while *)
figure = Graphics[
  GraphicsComplex[pts,
   {
    {White, AbsoluteThickness[0.3], Opacity[0.05],
     Line[List @@@ EdgeList[lg]]},
    {AbsolutePointSize[2], Opacity[0.7],
     MapIndexed[
      {ColorData[45]@First[#2], Point[#1]} &,
      c["Communities"]
     ]}
   }
  ],
  Background → Black
 ]
```

SHIFT-ENTER evaluation is disabled in the cell above to avoid running it accidentally. Running the code takes about 2-3 minutes on a modern computer. Copy the code to a new cell to try it.

## Gallery

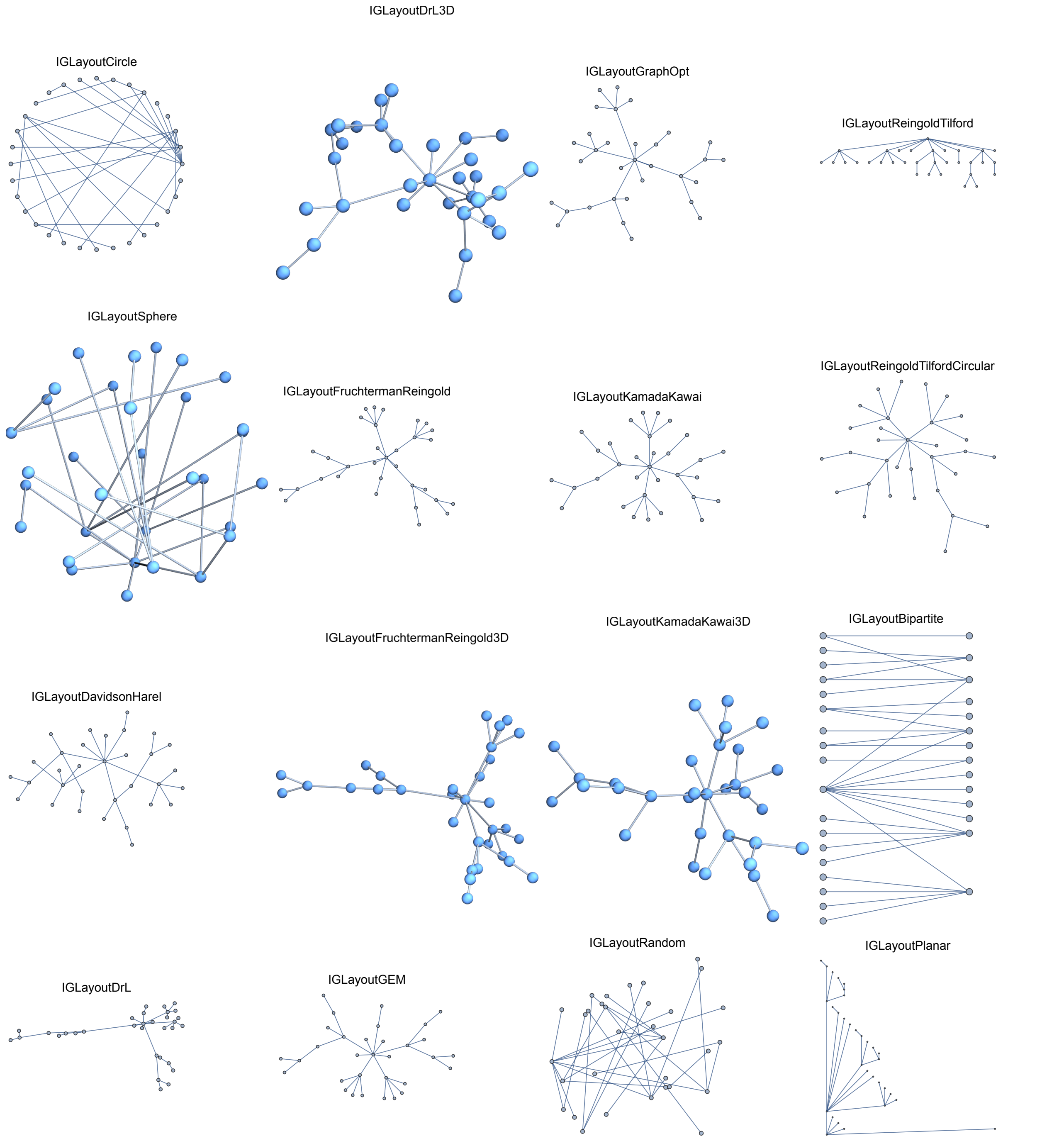
Create galleries of the various graph layouts available in IGraph/M.

Visualise a tree graph with all layouts.

In[972]:=

```
g = IGBarabasiAlbertGame[32, 1, DirectedEdges → False];
layouts = Graph[#[g], PlotLabel → #, LabelStyle → 7] & /@
   {IGLayoutCircle, IGLayoutSphere, IGLayoutDavidsonHarel, IGLayoutDrL, IGLayoutDrL3D,
    IGLayoutFruchtermanReingold, IGLayoutFruchtermanReingold3D, IGLayoutGEM, IGLayoutGraphOpt,
    IGLayoutKamadaKawai, IGLayoutKamadaKawai3D, IGLayoutRandom, IGLayoutReingoldTilford,
    IGLayoutReingoldTilfordCircular, IGLayoutBipartite, IGLayoutPlanar};
Multicolumn[layouts]
```

Out[974]=

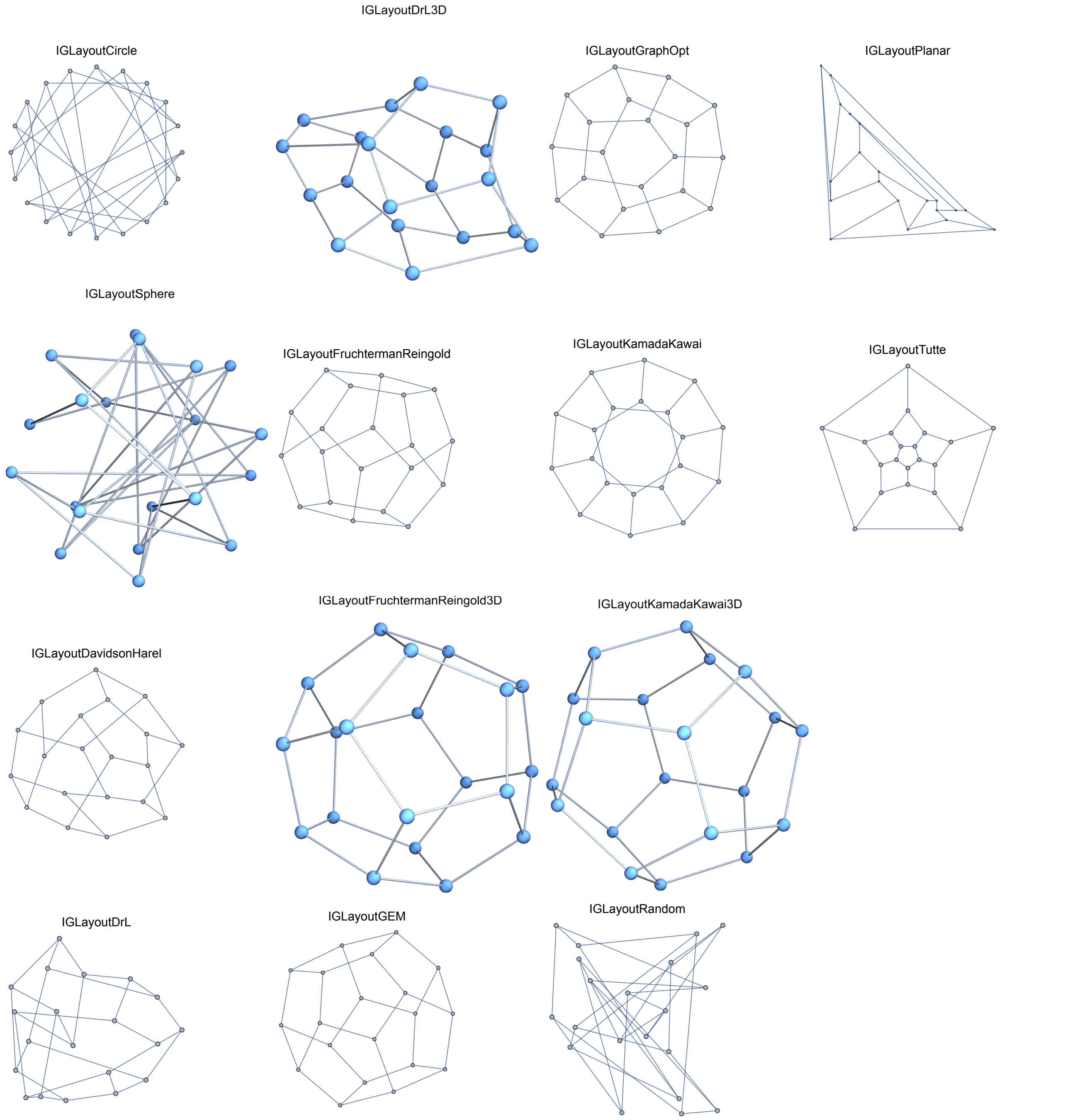


Visualise a polyhedral graph with all layouts.

In[975]:=

```
g = GraphData["DodecahedralGraph"];
layouts = Graph[#[g], PlotLabel → #, LabelStyle → 7] & /@
   {IGLayoutCircle, IGLayoutSphere, IGLayoutDavidsonHarel, IGLayoutDrL, IGLayoutDrL3D,
    IGLayoutFruchtermanReingold, IGLayoutFruchtermanReingold3D, IGLayoutGEM, IGLayoutGraphOpt,
    IGLayoutKamadaKawai, IGLayoutKamadaKawai3D, IGLayoutRandom, IGLayoutPlanar, IGLayoutTutte};
Multicolumn[layouts]
```

Out[977]=



# Community detection

The following functions are available:

In[978]:=

```
? IGCommunities*
```

▼IGraphM`

| | | |
|---|---|---|
| IGCommunitiesEdgeBetweenness | IGCommunitiesLabelPropagation | IGCommunitiesOptimalModularity |
| IGCommunitiesFluid | IGCommunitiesLeadingEigenvector | IGCommunitiesSpinGlass |
| IGCommunitiesGreedy | IGCommunitiesLeiden | IGCommunitiesWalktrap |
| IGCommunitiesInfoMAP | IGCommunitiesMultilevel | |

## Concepts

*Modularity* is defined for a given partitioning of a graph's vertices into *communities*. For undirected graphs, it is defined as

$$Q = \frac{1}{2\,m} \sum_{i,j} \left( A_{ij} - \frac{k_i\,k_j}{2\,m} \right) \delta_{c_i c_j},$$

where $m$ is the number of edges, $A$ is the adjacency matrix, $k_i$ is the degree of node $i$, and $c_i$ is the community that node $i$ belongs to. $\delta_{ij}$ is the Kronecker $\delta$ symbol. For weighted graphs, $A$ is the weighted adjacency matrix, $k_i$ are the sum of weights of edges incident on node $i$, and $m$ is the sum of all weights.

Modularity characterizes the tendency of vertices to connect more within their own group than with other groups, relative to a null model that considers vertex degrees to be fixed. For a given partitioning, it can be computed using `IGModularity`. Most community detection methods aim find a partitioning of the graph which results in high modularity.

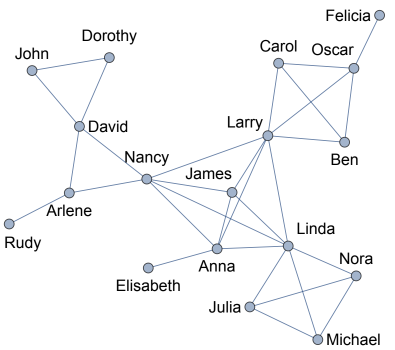
## Basic usage and utility functions

Community detection functions return `IGClusterData` objects.

In[979]:=

```
g = ExampleData[{"NetworkGraph", "FamilyGathering"}]
```

Out[979]=




In[980]:=

```
cl = IGCommunitiesGreedy[g]
```

Out[980]=

```
IGClusterData[ Elements: 18 Communities: 4 ]
```

The data available in the object can be queried using `IGClusterData[...]["Properties"]`. See the Examples section below for more information. In *Mathematica* 12.0 and later, Information can be used to get a quick human-readable summary.

In[981]:=

```
cl["Properties"]
```

Out[981]=

```
{Algorithm, Communities, ElementCount, Elements,
 HierarchicalClusters, Merges, Modularity, Properties, Tree}
```

In[982]:=

```
cl["Communities"]
```

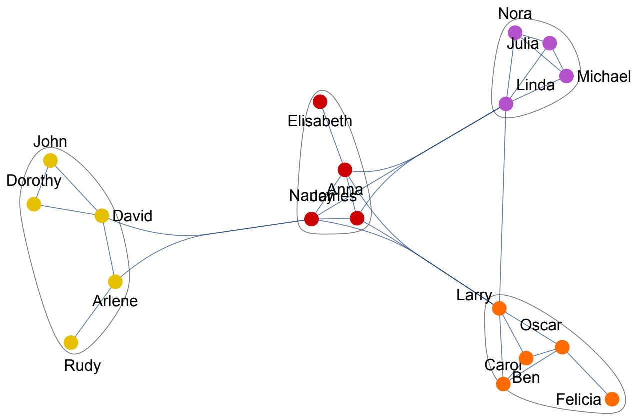
Out[982]=

```
{{Elisabeth, James, Anna, Nancy}, {John, Dorothy, David, Arlene, Rudy},
 {Linda, Michael, Nora, Julia}, {Larry, Carol, Ben, Oscar, Felicia}}
```

In[983]:=

```
CommunityGraphPlot[g, cl["Communities"], ImageSize → Medium]
```

Out[983]=




In[984]:=

```
IGModularity[g, cl]
```

Out[984]=

```
0.454735
```

## IGClusterData

In[985]:=

```
? IGClusterData
```

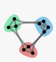
IGClusterData[association] represents the output of community detection functions. Properties can be queried using IGClusterData[…]["property"].

`IGClusterData` represents a partitioning of a graph into communities. This object cannot be created directly. It is returned by community detection functions. See the Examples section below for more information.

In[986]:=

```
cl = IGCommunitiesLabelPropagation@ExampleData[{"NetworkGraph", "FamilyGathering"}]
```

Out[986]=

IGClusterData[  Elements: 18 Communities: 3 ]

Query the available properties.

In[987]:=

```
cl["Properties"]
```

Out[987]=

```
{Algorithm, Communities, ElementCount, Elements, Modularity, Properties}
```

Retrieve the communities.

In[988]:=

```
cl["Communities"]
```

Out[988]=

```
{{Elisabeth, James, Anna, Linda, Larry, Carol, Nancy, David, Ben, Oscar, Felicia, Arlene, Rudy},
 {John, Dorothy}, {Michael, Nora, Julia}}
```

When the `"Modularity"` property is available, `Max[cl["Modularity"]]` gives the modularity of the current partitioning.

In[989]:=

```
Max[cl["Modularity"]]
```

Out[989]=

```
0.188866
```

## IGModularity

In[990]:=

```
? IGModularity
```

IGModularity[graph, {{v11, v12, …}, {v21, v22, …}, …}] gives the modularity
  the specified partitioning of graph's vertices into communities. Edge directions are ignored.
IGModularity[graph, clusterdata] uses the partitioning specified by an IGClusterData object.

`IGModularity` computes the generalized modularity in undirected or directed graphs, taking weights into account. For undirected graphs, it is defined as

$$Q = \frac{1}{2\,m} \sum_{i,j} \left( A_{ij} - \gamma \frac{k_i\, k_j}{2\,m} \right) \delta_{c_i c_j},$$

where $m$ is the number of edges, $A$ is the adjacency matrix, $k_i$ is the degree of node $i$, and $c_i$ is the community that node $i$ belongs to. $\delta_{ij}$ is the Kronecker $\delta$ symbol. For weighted graphs, $A$ is the weighted adjacency matrix, $k_i$ are the sum of weights of edges incident on node $i$, and $m$ is the sum of all weights. In the undirected case, $A_{ii}$ is assumed to contain twice the number of self-loops on vertex $i$, or twice their total weight in the weighted case. $\gamma$ is a resolution parameter, with $\gamma = 1$ yielding the standard modularity.

The directed generalization is

$$Q = \frac{1}{m} \sum_{i,j} \left( A_{ij} - \gamma \frac{k_i^{\text{out}}\, k_j^{\text{in}}}{m} \right) \delta_{c_i c_j}.$$

For simple graphs, `IGModularity[graph, communities]` is equivalent to `GraphAssortativity[graph, communities, "Normalized" → False]`. However, in contrast to `GraphAssortativity`, `IGModularity` does take self-loops and multi-edges into account.

Available options are:

- `"Resolution" → γ` sets the resolution parameter. The default is 1.
- `DirectedEdges → True` takes into account edge directions in directed graphs. By default, they are ignored.

Compute the modularity of a stochastic block model graph having three partitions, each with 20 vertices.

In[991]:=

```
g = IGStochasticBlockModelGame[(0.1    0.01  0.015
                                0.01   0.15  0.01
                                0.015  0.01  0.2), {20, 20, 20}];
```

In[992]:=

```
IGModularity[g, Partition[VertexList[g], 20]]
```

Out[992]=

```
0.456154
```

Setting the resolution parameter to zero computes the fraction of intra-community edges.

In[993]:=

```
IGModularity[g, Partition[VertexList[g], 20], "Resolution" → 0]
```

Out[993]=

```
0.8125
```

## IGModularityMatrix

In[994]:=

```
? IGModularityMatrix
```

IGModularityMatrix[graph] gives the modularity matrix of graph.

`IGModularityMatrix` computes the generalized modularity matrix of a graph, defined as

$$B_{ij} = A_{ij} - \gamma \frac{k_i k_j}{2m}$$

for undirected graphs and as

$$B_{ij} = A_{ij} - \gamma \frac{k_i^{\text{out}} k_j^{\text{in}}}{m}$$

for directed ones. Here, $A$ represents the adjacency matrix, $k_i$ is the degree of vertex $i$, $m$ is the sum of edge weights and $\gamma$ is the resolution parameter. Just as with `IGModularity`, in undirected graphs $A_{ii}$ is assumed to contain twice the number of self-loops. This way, the result for an undirected graph is the same as for the corresponding directed one where all undirected edges are replaced by a pair of reciprocal directed ones.

Available options are:

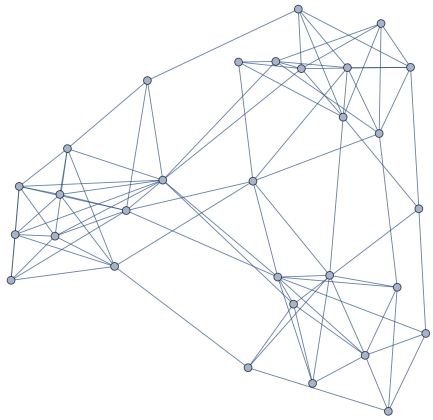
- `"Resolution"` → $\gamma$ sets the resolution parameter. The default is 1.
- `DirectedEdges` → `True` takes into account edge directions in directed graphs. By default, they are ignored.

The modularity matrix is used in spectral clustering algorithms, such as the one implemented by `IGCommunitiesLeadingEigenvector`. Communities can be separated based on spatial clustering of the points formed by the first few eigenvectors of the modularity matrix.

In[995]:=

```
g = IGStochasticBlockModelGame[0.05 + 0.5 IdentityMatrix[3], {10, 10, 10}]
```

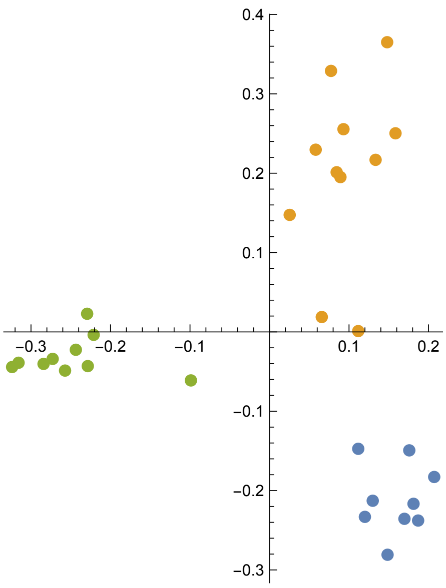
Out[995]=



In[996]:=

```
ListPlot[FindClusters@Transpose@Eigenvectors[IGModularityMatrix[g], 2],
 PlotStyle → PointSize[Large], AspectRatio → Automatic]
```

Out[996]=



## IGCompareCommunities

In[997]:=

```
? IGCompareCommunities
```

IGCompareCommunities[clusterdata1, clusterdata2] compares two community structures given as IGClusterData objects using all available methods. Available methods: {"VariationOfInformation", "NormalizedMutualInformation", "SplitJoinDistance", "UnadjustedRandIndex", "AdjustedRandIndex"}.
IGCompareCommunities[clusterdata1, clusterdata2, method] compares two community structures using method.
IGCompareCommunities[clusterdata1, clusterdata2, {method1, …}] compares two community structures using each given method.
IGCompareCommunities[graph, communities1, communities2] compares two partitionings of the graph vertices into communities using all available methods.
IGCompareCommunities[graph, communities1, communities2, method] compares two community structures using method.
IGCompareCommunities[graph, communities1, communities2, {method1, …}] compares two community structures using each given method.
IGCompareCommunities[vertexList, communities1, communities2] uses the given vertex list.

Some of these measures are defined based on the entropy of a discrete random variable associated with a given clustering $C$ of vertices. Let $p_i$ be the probability that a randomly picked vertex would be part of cluster $i$. Then the entropy of the clustering is

$$H(C) = -\sum_i p_i \ln p_i.$$

Similarly, we can define the joint entropy of two clusterings $C_1$ and $C_2$ based on the probability $p_{ij}$ that a random vertex is part of cluster $i$ in the first clustering and cluster $j$ in the second one:

$$H(C_1, C_2) = -\sum_{i,j} p_{ij} \ln p_{ij}.$$

The **mutual information** of $C_1$ and $C_2$ is then $\mathrm{MI}(C_1, C_2) = H(C_1) + H(C_2) - H(C_1, C_2) \geq 0$. A large mutual information indicates a high overlap between the two clusterings. The **normalized mutual information**, as computed by igraph, is

$$\mathrm{NMI}(C_1, C_2) = \frac{2\,\mathrm{MI}(C_1, C_2)}{H(C_1) + H(C_2)}.$$

It takes its value from the interval (0, 1], with 1 achieved when the two clusterings coincide.

The **variation of information** is defined as $\mathrm{VI}(C_1, C_2) = [H(C_1) - \mathrm{MI}(C_1, C_2)] + [H(C_2) - \mathrm{MI}(C_1, C_2)] = 2\,H(C_1, C_2) - H(C_1) - H(C_2)$. Lower values of the variation of information indicate a smaller difference between the two clusterings, with $\mathrm{VI} = 0$ achieved precisely when they coincide.

The **Rand index** is defined based on counting pairs of vertices that are within the same or different clusters in the two clusterings. Let $a_1$ and $a_2$ denote the number of pairs that are grouped together in $C_1$ and $C_2$, respectively. Then the Rand index is

$$\mathrm{RI}(C_1, C_2) = \frac{a_1 + a_2}{\binom{n}{2}},$$

where $\binom{n}{2}$ is the total number of pairs of $n$ vertices. The value of the Rand index varies between 0 and 1. The **adjusted Rand index**, ARI, corrects the value based on the Rand index expected after after a random rearrangement of the vertices, denoted ERI:

$$\mathrm{ARI} = \frac{\mathrm{RI} - \mathrm{ERI}}{1 - \mathrm{ERI}}.$$

In[998]:=
```
g = ExampleData[{"NetworkGraph", "FamilyGathering"}];
```
In[999]:=
```
{cl1, cl2} = {IGCommunitiesGreedy[g], IGCommunitiesEdgeBetweenness[g]}
```
Out[999]=
```
{IGClusterData[ Elements: 18 Communities: 4 ], IGClusterData[ Elements: 18 Communities: 4 ]}
```
In[1000]:=
```
IGCompareCommunities[cl1, cl2]
```
Out[1000]=
```
<|VariationOfInformation → 0.278001, NormalizedMutualInformation → 0.899283,
 SplitJoinDistance → 2, UnadjustedRandIndex → 0.947712, AdjustedRandIndex → 0.841942|>
```

## Community detection methods

### IGCommunitiesEdgeBetweenness

In[1001]:=
```
? IGCommunitiesEdgeBetweenness
```

IGCommunitiesEdgeBetweenness[graph] finds communities using the Girvan–Newman algorithm.

`IGCommunitiesEdgeBetweenness[]` implements the Girvan–Newman algorithm.

Weighted graphs are supported. Weights are treated as “distances”, i.e. a large weight represents a weak connection.

Available option values:

- `"ClusterCount"`, the number of communities to return. Default: `Automatic`.

Special properties returned with the result:

- `"RemovedEdges"` is the list of edges removed in each step of the algorithm.
- `"Bridges"` records the steps which resulted in splitting the graph into more components.

#### References

- M. Girvan and M. E. J. Newman: Community structure in social and biological networks, *PNAS* 99, 7821-7826 (2002).

### IGCommunitiesFluid

In[1002]:=
```
? IGCommunitiesFluid
```

IGCommunitiesFluid[graph, clusterCount] finds communities using the fluid communities algorithm.

`IGCommunitiesFluid[]` implements the fluid communities algorithm.

#### Reference

- F. Parés, D. Garcia-Gasulla, A. Vilalta, J. Moreno, E. Ayguadé, Jesús Labarta, U. Cortés, T. Suzumura: Fluid Communities: A Competitive, Scalable and Diverse Community Detection Algorithm, https://arxiv.org/abs/1703.09307

## IGCommunitiesGreedy

In[1003]:=

```
? IGCommunitiesGreedy
```

IGCommunitiesGreedy[graph] finds communities using greedy optimization of modularity.

`IGCommunitiesGreedy[]` implements greedy optimization of modularity.

Weighted graphs are supported.

### Reference


- A. Clauset, M. E. J. Newman, C. Moore: Finding community structure in very large networks, http://www.arxiv.org/abs/cond-mat/0408187


## IGCommunitiesInfoMAP

In[1004]:=

```
? IGCommunitiesInfoMAP
```

IGCommunitiesInfoMAP[graph] finds communities using the InfoMAP algorithm. The default number of trials is 10.
IGCommunitiesInfoMAP[graph, trials]

`IGCommunitiesInfoMAP[]` implements the InfoMAP algorithm.

It supports both edge weights and vertex weights.

The default number of trials is 10.

Special properties returned with the result:

- `"CodeLength"` is the code length of the partition.

### References


- M. Rosvall and C. T. Bergstrom, Maps of information flow reveal community structure in complex networks, *PNAS* 105, 1118 (2008)
- M. Rosvall, D. Axelsson, and C. T. Bergstrom, The map equation, *Eur. Phys. J. Special Topics* 178, 13 (2009)


## IGCommunitiesLabelPropagation

In[1005]:=

```
? IGCommunitiesLabelPropagation
```

IGCommunitiesLabelPropagation[graph] finds communities by assigning labels
to each vertex and then updating them by majority voting in the neighbourhood of the vertex.

Weighted graphs are supported.

### References


- Raghavan, U.N. and Albert, R. and Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76, 036106. (2007).

## IGCommunitiesLeadingEigenvector

In[1006]:=

```
? IGCommunitiesLeadingEigenvector
```

IGCommunitiesLeadingEigenvector[graph] finds communities based on the leading eigenvector of the modularity matrix.

Weighted graphs are supported.

Available option values:

- `"ClusterCount"`, the number of communities to return. May return fewer communities than requested. Default: `Automatic`.

### References

- M. E. J. Newman: Finding community structure using the eigenvectors of matrices, *Phys. Rev. E* 74:036104 (2006).

## IGCommunitiesMultilevel

In[1007]:=

```
? IGCommunitiesMultilevel
```

IGCommunitiesMultilevel[graph] finds communities using the Louvain method.

`IGCommunitiesMultilevel[]` implements the Louvain community detection method.

Weighted graphs are supported.

### References

- V. D. Blondel, J.-L. Guillaume, R. Lambiotte and E. Lefebvre: Fast unfolding of community hierarchies in large networks, *J. Stat. Mech.* P10008 (2008)

## IGCommunitiesLeiden

In[1008]:=

```
? IGCommunitiesLeiden
```

IGCommunitiesLeiden[graph] finds communities using the Leiden method.

The Leiden algorithm is similar to the multilevel algorithm, often called the Louvain algorithm, but it is faster and yields higher quality solutions. It can optimize both modularity and the Constant Potts Model, which does not suffer from the resolution-limit (see preprint http://arxiv.org/abs/1104.3083).

The Leiden algorithm consists of three phases: (1) local moving of nodes, (2) refinement of the partition and (3) aggregation of the network based on the refined partition, using the non-refined partition to create an initial partition for the aggregate network. In the local move procedure in the Leiden algorithm, only nodes whose neighborhood has changed are visited. The refinement is done by restarting from a singleton partition within each cluster and gradually merging the subclusters. When aggregating, a single cluster may then be represented by several nodes (which are the subclusters identified in the refinement).

The Leiden algorithm provides several guarantees. The Leiden algorithm is typically iterated: the output of one iteration is used as the input for the next iteration. At each iteration all clusters are guaranteed to be connected and well-separated. After an iteration in which nothing has changed, all nodes and some parts are guaranteed to be locally optimally assigned. Finally, asymptotically, all subsets of all clusters are guaranteed to be locally optimally assigned.

The Leiden method maximizes a quality measure (a generalization of modularity) defined as

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \gamma n_i n_j) \delta_{c_i c_j}$$

where $m$ is the sum of edge weights (number of edges if the graph is unweighted), $A$ is the weighted adjacency matrix, $n_i$ is the weight of vertex $i$, and $c_i$ is the community that vertex $i$ belongs to. $\delta_{ij}$ is the Kronecker $\delta$ symbol.

$\gamma$ is a resolution parameter that can be set with the "Resolution" option.

The function chooses the vertex weights automatically, according to the value of the VertexWeight option:

- VertexWeight → "NormalizedStrength" (default) sets $n_i = k_i \big/ \sqrt{2m}$, where $k_i$ is the strength (sum of incident edge weights) of vertex $i$. If $\gamma = 1$, then the quality measure becomes equivalent to the modularity.
- VertexWeight → "Constant" sets $n_i = 1$. With this choice, it is recommended to set the resolution parameter $\gamma$ explicitly. A reasonable $\gamma$ value for unweighted graphs is the graph density.
- VertexWeight → "VertexWeight" takes vertex weights from the VertexWeight graph property.

Other available options:

- "Resolution" → $\gamma$ sets the resolution parameter $\gamma$. The default is $\gamma = 1$. With VertexWeight → "NormalizedStrength", a reasonable value is 1. With VertexWeight → "Constant", a reasonable value is the graph density.
- "Beta" → $\beta$ sets the randomness used in the refinement step when merging clusters. The default is $\beta = 0.01$.

Special properties returned with the result :

- "Quality" is the value of the quality measure $Q$.

Examples:

In[1009]:=
```
g =
  Graph[ExampleData[{"NetworkGraph", "LesMiserables"}], GraphStyle → "BasicBlack", VertexSize → 2];
```

With the default option values VertexWeight → "NormalizedStrength" and "Resolution" → 1, IGCommunitiesLeiden effectively uses the modularity as the quality measure.

In[1010]:=
```
cl = IGCommunitiesLeiden[g]
```

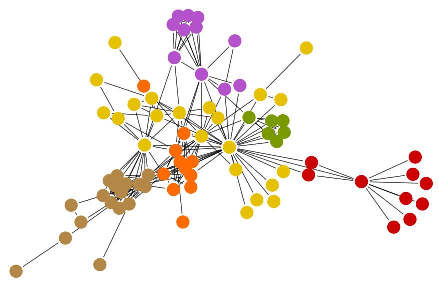
Out[1010]=
```
IGClusterData[ Elements: 77  Communities: 6 ]
```

In[1011]:=
```
{cl["Quality"], IGModularity[g, cl]}
```

Out[1011]=
```
{0.566688, 0.566688}
```

In[1012]:=
```
HighlightGraph[g, cl["Communities"]]
```

Out[1012]=

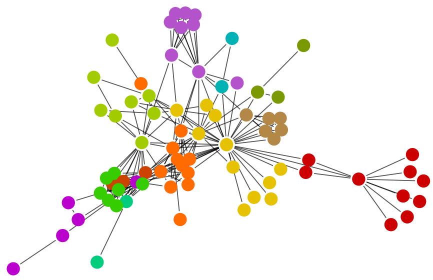


A higher "Resolution" value results in more communities.

In[1013]:=

```
HighlightGraph[
 g,
 IGCommunitiesLeiden[g, "Resolution" → 3]["Communities"]
]
```

Out[1013]=

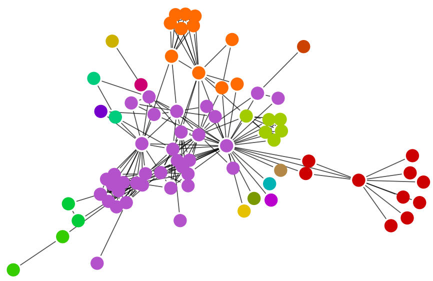


With VertexWeight → "Constant", it is recommended to set "Resolution" explicitly. A reasonable starting point is GraphDensity[g].

In[1014]:=

```
HighlightGraph[
 g,
 IGCommunitiesLeiden[g, VertexWeight → "Constant", "Resolution" → 0.1]["Communities"]
]
```

Out[1014]=



### References


- Traag, V. A., Waltman, L., van Eck, N. J. (2019). From Louvain to Leiden: guaranteeing well-connected communities. Scientific Reports, 9(1), 5233. http://dx.doi.org/10.1038/s41598-019-41695-z


## IGCommunitiesOptimalModularity

In[1015]:=

```
? IGCommunitiesOptimalModularity
```

IGCommunitiesOptimalModularity[graph] finds communities by maximizing the modularity through integer programming.

Finds the clustering that maximizes modularity exactly. This algorithm is very slow.

Weighted graphs are supported.

## IGCommunitiesSpinGlass

In[1016]:=

```
? IGCommunitiesSpinGlass
```

IGCommunitiesSpinGlass[graph] finds communities using a spin glass model and simulated annealing. Available "UpdateRule" option values: {"Simple", "Configuration"}. Available Method options: {"Original", "Negative"}.

Weighted graphs are supported.

Option values for `Method` are:

- `"Original"` only supports positive edge weights, but doesn't check that the supplied weights are actually positive.
- `"Negative"` supports negative weights as well.
- `Automatic` selects `"Negative"` if negative weights are presents and `"Original"` otherwise.

Option values for `"UpdateRule"` are: `"Simple"`, `"Configuration"`

Special properties returned with the result:

- `"FinalTemperature"` is the final temperature at the end of the algorithm.

### References

- For `Method → "Original`, see Joerg Reichardt and Stefan Bornholdt: Statistical Mechanics of Community Detection, http://arxiv.org/abs/cond-mat/0603718
- For `Method → "Negative"`, see V. A. Traag and Jeroen Bruggeman: Community detection in networks with positive and negative links, http://arxiv.org/abs/0811.2329

## IGCommunitiesWalktrap

In[1017]:=

```
? IGCommunitiesWalktrap
```

IGCommunitiesWalktrap[graph] finds communities via short random walks (of length 4 by default).
IGCommunitiesWalktrap[graph, steps] finds communities via random walks of length steps.

`IGCommunitiesWalktrap[]` finds communities using short random walks, exploiting the fact that random walks tend to stay within the same cluster.

Weighted graphs are supported.

The default number of steps is 4.

Available option values:

- `"ClusterCount"`, the number of communities to return. Default: `Automatic`.

### References

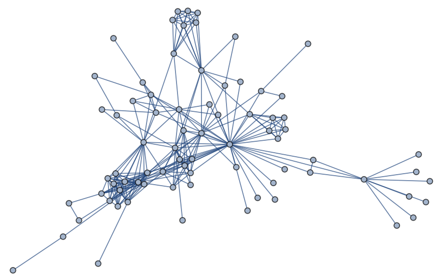
- Pascal Pons, Matthieu Latapy: Computing communities in large networks using random walks, http://arxiv.org/abs/physics/0512106

## Examples

In[1018]:=

```
g = ExampleData[{"NetworkGraph", "LesMiserables"}]
```

Out[1018]=



In[1019]:=

```
IGEdgeWeightedQ[g]
```

Out[1019]=

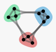
```
True
```

Community detection functions return `IGClusterData` objects.

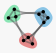
In[1020]:=

```
cl1 = IGCommunitiesEdgeBetweenness[g, "ClusterCount" → 7]
cl2 = IGCommunitiesWalktrap[g]
```

Out[1020]=

IGClusterData[  Elements: 77 Communities: 7 ]

Out[1021]=

IGClusterData[  Elements: 77 Communities: 9 ]

Various properties of these objects can be queried:

In[1022]:=

```
cl1["Communities"]
```

Out[1022]=

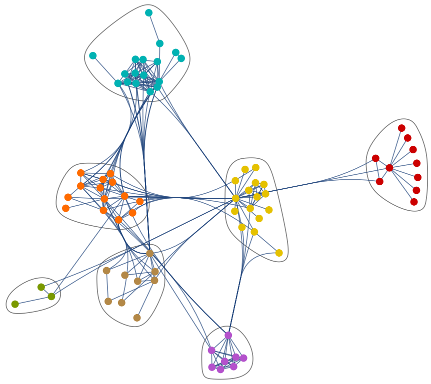
```
{{Myriel, Napoleon, Mlle Baptistine, Mme. Magloire,
  Countess De Lo, Geborand, Champtercier, Cravatte, Count, Old Man},
 {Labarre, Valjean, Marguerite, Mme. De R, Isabeau, Gervais, Bamatabois, Perpetue,
  Simplice, Scaufflaire, Woman1, Judge, Champmathieu, Brevet, Chenildieu, Cochepaille},
 {Tholomyes, Listolier, Fameuil, Blacheville, Favourite, Dahlia, Zephine, Fantine},
 {Mme. Thenardier, Thenardier, Cosette, Javert, Boulatruelle, Eponine,
  Anzelma, Woman2, Gueulemer, Babet, Claquesous, Montparnasse, Toussaint, Brujon},
 {Fauchelevent, Mother Innocent, Gribier}, {Pontmercy, Gillenormand, Magnon,
  Mlle Gillenormand, Mme. Pontmercy, Mlle Vaubois, Lt. Gillenormand, Marius, Baroness T},
 {Jondrette, Mme. Burgon, Gavroche, Mabeuf, Enjolras, Combeferre, Prouvaire, Feuilly,
  Courfeyrac, Bahorel, Bossuet, Joly, Grantaire, Mother Plutarch, Child1, Child2, Mme. Hucheloup}}
```

Visualize the detected communities in two different ways:

In[1023]:=

```
CommunityGraphPlot[g, cl1["Communities"]]
```

Out[1023]=



In[1024]:=

```
HighlightGraph[g, Subgraph[g, #] & /@ cl1["Communities"], GraphHighlightStyle → "DehighlightGray"]
```

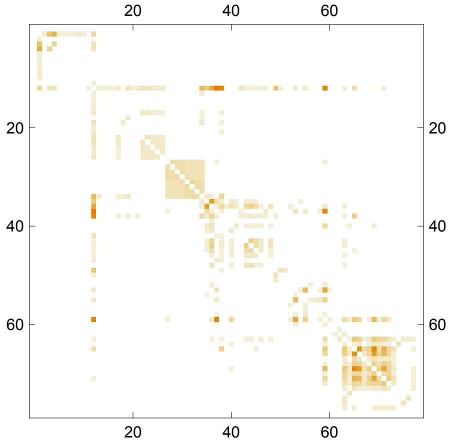
Out[1024]=

Plot the adjacency matrix, reordered to show the community structure.

In[1025]:=

```
IGAdjacencyMatrixPlot[g, Catenate@cl1["Communities"]]
```

Out[1025]=




The available properties depend on which algorithm was used for community detection. The following are always present:

- "Properties" returns all available properties.
- "Algorithm" returns the algorithm used for community detection.
- "Communities" returns the list of communities.
- "Elements" returns the vertices of the graph.
- "ElementCount" returns the vertex count of the graph.

These are present for hierarchical clustering methods:

- "HierarchicalClusters" returns the clustering in a format compatible with the Hierarchical Clustering standard package. **Note:** Isolated vertices may not be included.
- "Merges" represents the hierarchical clustering as a sequence of element merges. Elements are represented by their integer indices, and higher indices are introduced for the subclusters formed by the merges. This format is similar to the one used by MATLAB and many other tools. **Note:** Isolated vertices may not be included.
- "Tree" gives a binary tree representation of the merges. **Note:** Isolated vertices may not be included.

Additionally, the following, and other, algorithm-specific properties may be present:

- "Modularity" is a list of modularities for each step of the algorithm, or a single-element list containing the modularity corresponding to the returned clustering. What constitutes a step depends on the particular algorithm.

The "RemovedEdges" property is specific to the "EdgeBetweenness" method, and isn't present for "Walktrap".

In[1026]:=
```
cl1["Properties"]
```
Out[1026]=
```
{Algorithm, Bridges, Communities, EdgeBetweenness, ElementCount,
 Elements, HierarchicalClusters, Merges, Properties, RemovedEdges, Tree}
```
In[1027]:=
```
Take[cl1["RemovedEdges"], 10]
```
Out[1027]=
```
{Valjean ⟷ Myriel, Valjean ⟷ Mlle Baptistine, Valjean ⟷ Mme. Magloire,
 Gavroche ⟷ Valjean, Gavroche ⟷ Javert, Thenardier ⟷ Fantine, Bamatabois ⟷ Javert,
 Bossuet ⟷ Valjean, Montparnasse ⟷ Valjean, Gueulemer ⟷ Gavroche}
```
In[1028]:=
```
cl2["Properties"]
```
Out[1028]=
```
{Algorithm, Communities, ElementCount, Elements,
 HierarchicalClusters, Merges, Modularity, Properties, Tree}
```
Multiple properties may be retrieved at the same time.

In[1029]:=
```
cl2[{"Algorithm", "ElementCount"}]
```
Out[1029]=
```
{Walktrap, 77}
```
Compare the two clusterings:

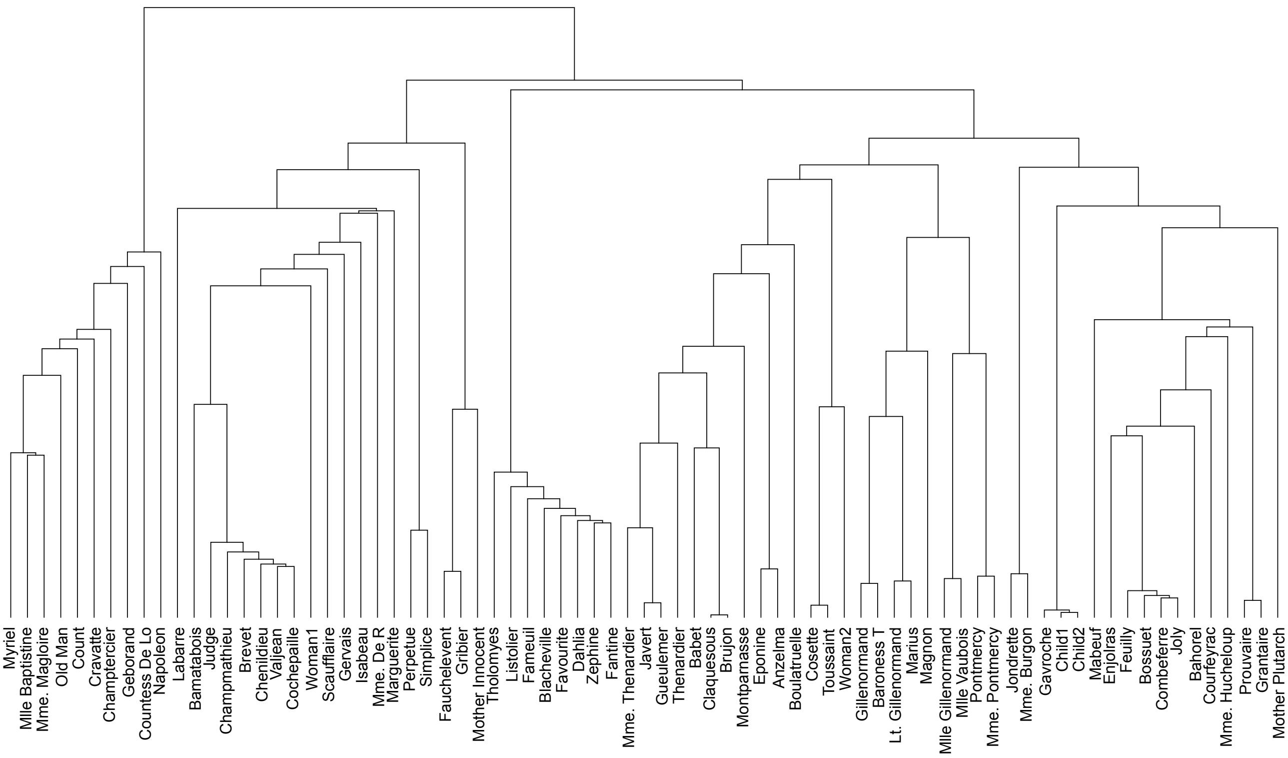
In[1030]:=
```
IGCompareCommunities[cl1, cl2]
```
Out[1030]=
```
<|VariationOfInformation → 0.804544, NormalizedMutualInformation → 0.786844,
 SplitJoinDistance → 29, UnadjustedRandIndex → 0.879699, AdjustedRandIndex → 0.555464|>
```
Visualize the hierarchical clustering using the Hierarchical Clustering Package.

In[1031]:=
```
<< HierarchicalClustering`
```

In[1032]:=

```
DendrogramPlot[cl1["HierarchicalClusters"],
 LeafLabels → (Rotate[#, Pi / 2] &), ImageSize → 750, AspectRatio → 1 / 2]
```

Out[1032]=




Hierarchical community structures can also be obtained as a vertex-weighted tree graph.

In[1033]:=

```
g = ExampleData[{"NetworkGraph", "ZacharyKarateClub"}];
```

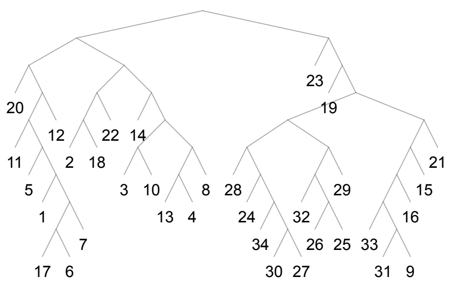
In[1034]:=

```
cl = IGCommunitiesGreedy[g];
```

In[1035]:=

```
clusteringTree = cl["Tree"]
```

Out[1035]=




In[1036]:=

```
{GraphQ[clusteringTree], IGVertexWeightedQ[clusteringTree]}
```

Out[1036]=

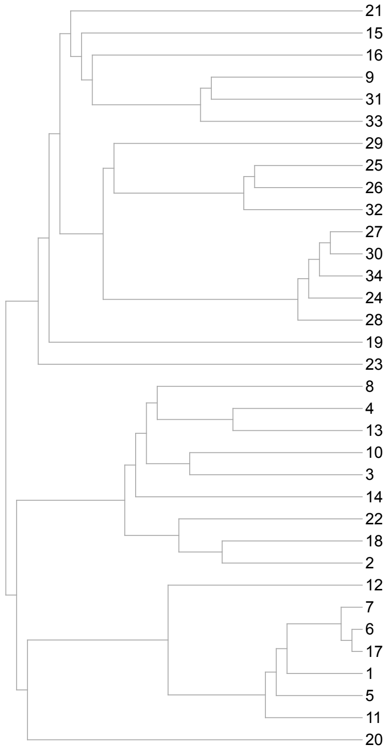
```
{True, True}
```

This tree can be supplied as input to `Dendrogram`.

In[1037]:=

```
Dendrogram[clusteringTree, Left]
```

Out[1037]=




# Graph cycles

## Eulerian paths and cycles

An Eulerian path passes through each edge of a graph precisely once. An Eulerian cycle is a closed Eulerian path: its starting vertex is the same as its ending vertex. Eulerian paths are also known as Eulerian trails.

**Note:** As of IGraph/M 0.5, the Eulerian path functions are still experimental.

### IGEulerianQ

In[1038]:=

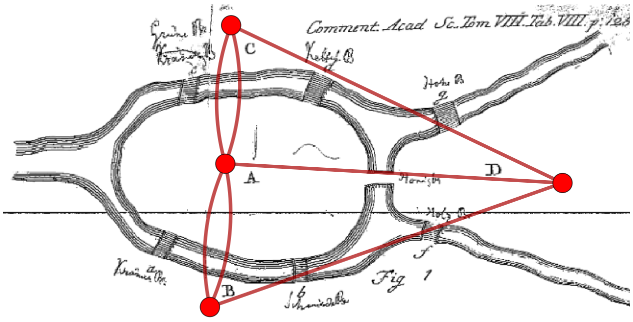
```
? IGEulerianQ
```

IGEulerianQ[graph] tests if graph has a path that traverses each edge once (Eulerian path).
IGEulerianQ[graph, Closed -> True] tests if graph has a cycle that traverses each edge once.

The following graph does not have an Eulerian path:

In[1039]:=

```
graph = 
```




```
;
```

In[1040]:=

```
IGEulerianQ[graph]
```

Out[1040]=

```
False
```

Removing the edge A ⟷ D makes it Eulerian:

In[1041]:=

```
IGEulerianQ@EdgeDelete[graph, "A" ↔ "D"]
```

Out[1041]=

```
True
```

But it will only have an Eulerian path, not an Eulerian cycle:

In[1042]:=

```
IGEulerianQ[
 EdgeDelete[graph, "A" ↔ "D"],
 Closed → True
]
```

Out[1042]=

```
False
```

One possible path is the following:

In[1043]:=

```
IGEulerianPathVertices[EdgeDelete[graph, "A" ↔ "D"]]
```

Out[1043]=

```
{B, A, B, D, C, A, C}
```

## IGEulerianPath and IGEulerianPathVertices

In[1044]:=

```
? IGEulerianPath
```

IGEulerianPath[graph] returns the edges of an Eulerian path, if it exists.
IGEulerianPath[graph, Closed -> True] returns an Eulerian cycle.

In[1045]:=

```
? IGEulerianPathVertices
```

IGEulerianPathVertices[graph] returns the vertices of an Eulerian path, if it exists.
IGEulerianPathVertices[graph, Closed -> True] returns the vertices of an Eulerian cycle.

Find an Eulerian cycle through an icosidodecahedral graph:

In[1046]:=

```
g = GraphData["IcosidodecahedralGraph"];
cycle = IGEulerianPath[g, Closed → True]
```

Out[1047]=

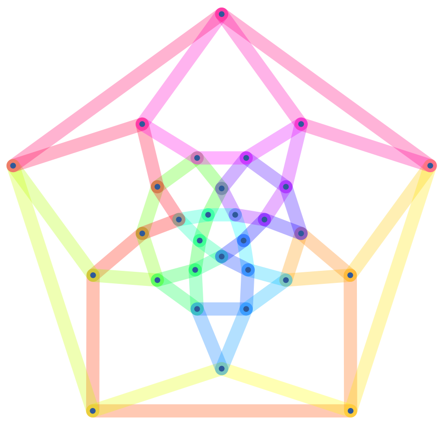
```
{1 ⟷ 5, 5 ⟷ 7, 7 ⟷ 8, 6 ⟷ 8, 2 ⟷ 6, 2 ⟷ 16, 6 ⟷ 16, 6 ⟷ 30, 8 ⟷ 30, 8 ⟷ 14, 7 ⟷ 14,
 7 ⟷ 29, 5 ⟷ 29, 5 ⟷ 15, 1 ⟷ 15, 1 ⟷ 18, 18 ⟷ 20, 11 ⟷ 20, 11 ⟷ 12, 3 ⟷ 12, 3 ⟷ 9,
 9 ⟷ 15, 15 ⟷ 22, 9 ⟷ 22, 9 ⟷ 17, 3 ⟷ 17, 3 ⟷ 27, 12 ⟷ 27, 12 ⟷ 13, 4 ⟷ 13, 4 ⟷ 10,
 10 ⟷ 16, 16 ⟷ 23, 14 ⟷ 23, 14 ⟷ 22, 22 ⟷ 23, 10 ⟷ 23, 10 ⟷ 17, 4 ⟷ 17, 4 ⟷ 28, 2 ⟷ 28,
 2 ⟷ 19, 19 ⟷ 21, 11 ⟷ 21, 11 ⟷ 13, 13 ⟷ 28, 19 ⟷ 28, 19 ⟷ 26, 21 ⟷ 26, 20 ⟷ 21,
 20 ⟷ 25, 24 ⟷ 25, 24 ⟷ 26, 26 ⟷ 30, 24 ⟷ 30, 24 ⟷ 29, 25 ⟷ 29, 18 ⟷ 25, 18 ⟷ 27, 1 ⟷ 27}
```

Visualize it using colour hues:

In[1048]:=

```
HighlightGraph[g,
 MapIndexed[Style[#1, Hue[First[#2] / Length[cycle]]] &, cycle],
 GraphStyle → "ThickEdge"
]
```

Out[1048]=

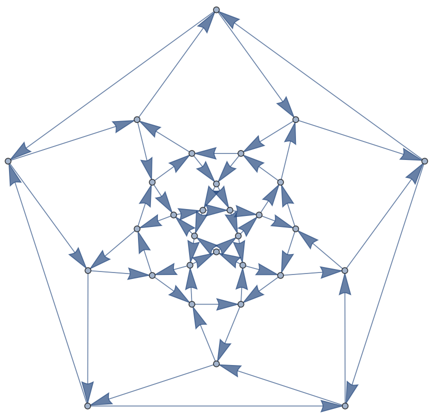


Direct the edges of the graph along the cycle:

In[1049]:=

```
Graph[
 VertexList[g],
 DirectedEdge @@@ Partition[IGEulerianPathVertices[g, Closed → True], 2, 1],
 EdgeStyle → Arrowheads[Medium],
 VertexCoordinates → GraphEmbedding[g]
]
```

Out[1049]=



# Graph colouring

The graph colouring problem is assigning “colours” or “labels” to the vertices of a graph so that no two adjacent vertices will have the same colour. Similarly, edge colouring assigns colours to edges so that adjacent edges never have the same colour.

IGraph/M represents colours with the integers `1, 2, …`. Edge directions and self-loops are ignored.

## Fast heuristic colouring

In[1050]:=

```
? IGVertexColoring
```

IGVertexColoring[graph] gives a vertex colouring of graph.

In[1051]:=

```
? IGEdgeColoring
```

IGEdgeColoring[graph] gives an edge colouring of graph.

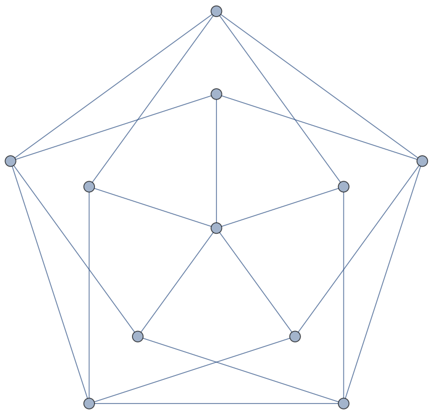
These function will find a colouring of the graph using a fast heuristic algorithm. The colouring may not be minimal. Edge directions are ignored.

Compute a vertex colouring of a Mycielski graph.

In[1052]:=

```
g = GraphData[{"Mycielski", 4}]
```

Out[1052]=



`IGVertexColoring` returns a list of integers, each representing the colour of the vertex that is in the same position in the vertex list.

In[1053]:=

```
IGVertexColoring[g]
```

Out[1053]=

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

Associate the colours with vertex names.

In[1054]:=

```
AssociationThread[VertexList[g], IGVertexColoring[g]]
```

Out[1054]=

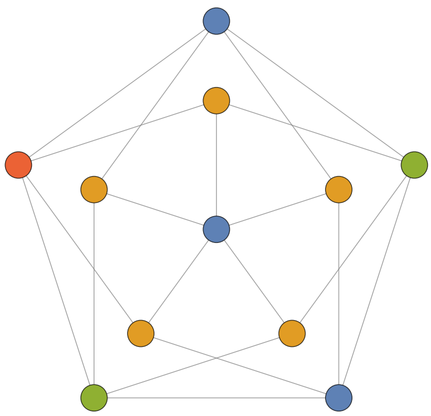
```
<|1 → 4, 2 → 3, 3 → 1, 4 → 1, 5 → 3, 6 → 1, 7 → 2, 8 → 2, 9 → 2, 10 → 2, 11 → 2|>
```

Visualize the colours using IGraph/M's property mapping functionality. See the *Property handling functions* documentation section for more information.

In[1055]:=

```
Graph[g, VertexSize → 1 / 3, EdgeStyle → Gray] //
 IGVertexMap[ColorData[97], VertexStyle → IGVertexColoring]
```

Out[1055]=

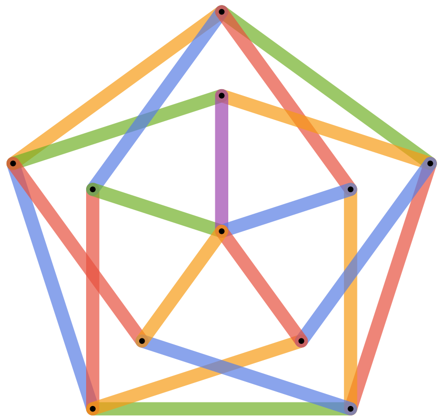


Visualize an edge colouring of the same graph.

In[1056]:=

```
Graph[g, GraphStyle → "ThickEdge", EdgeStyle → Opacity[0.7], VertexStyle → Black] //
 IGEdgeMap[ColorData[106], EdgeStyle → IGEdgeColoring]
```

Out[1056]=

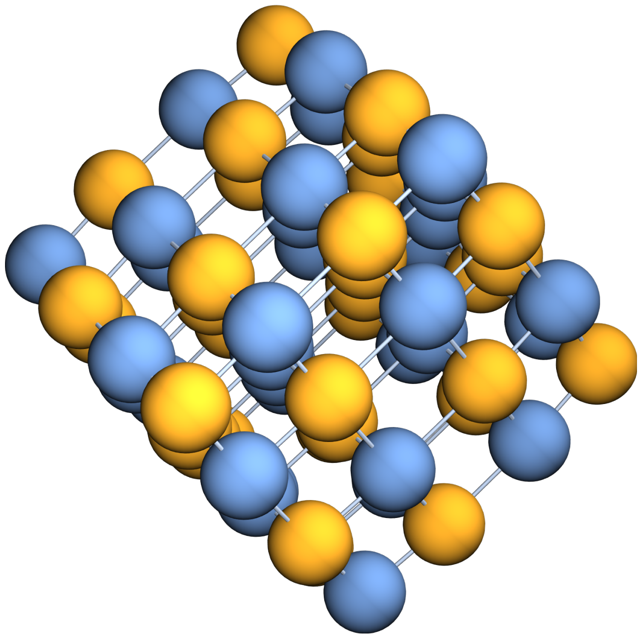


Compute a checkerboard-like colouring of a three-dimensional grid graph.

In[1057]:=

```
IGVertexMap[ColorData[97], VertexStyle → IGVertexColoring,
 Graph3D@GridGraph[{4, 4, 4}, VertexSize → 0.8]]
```

Out[1057]=



Compute a colouring of a Voronoi mesh.

In[1058]:=
```
mesh = VoronoiMesh[RandomReal[1, {20, 2}]]
```
Out[1058]=

In[1059]:=
```
col = IGVertexColoring@IGMeshCellAdjacencyGraph[mesh, 2]
```
Out[1059]=
```
{2, 5, 4, 2, 1, 1, 2, 2, 4, 1, 3, 1, 2, 3, 3, 4, 2, 1, 3, 1}
```
In[1060]:=
```
SetProperty[{mesh, {2, All}}, MeshCellStyle → ColorData[97] /@ col]
```
Out[1060]=

Compute a colouring of the map of African countries.

In[1061]:=
```
countries = CountryData["Africa"];
borderingQ[c1_, c2_] := MemberQ[c1["BorderingCountries"], c2]
graph = RelationGraph[borderingQ, countries];
```

In[1064]:=

```
GeoGraphics@MapThread[{GeoStyling[{Opacity[0.5], #2}], Polygon[#1]} &,
  {countries, ColorData[97] /@ IGVertexColoring[graph]}]
```

Out[1064]=


## *k*-colouring

In[1065]:=

```
? IGKVertexColoring
```

IGKVertexColoring[graph, k] attempts to find a k–colouring of graph's vertices. If none exist, {} is returned.

In[1066]:=

```
? IGKEdgeColoring
```

IGKEdgeColoring[graph, k] attempts to find a k–colouring of graph's edges. If none exist, {} is returned.

These functions find a colouring with *k* or fewer colours. They work by transforming the colouring into a satisfiability problem and using `SatisfiabilityInstances`.

The available option values are:

- `"ForcedColoring"` → `{v1, v2, …}` forces the given vertices to distinct and increasing colours. Normally, the vertices of a clique are given (which require as many colours as the size of the clique). The main purpose of this option is to reduce the number of redundant solutions of the equivalent SAT problem, and thus improve performance. When using edge colouring functions, a set of edges should be passed.
- `"ForcedColoring"` → `"MaxDegreeClique"` attempts to find a clique containing a maximum degree vertex, and forces colours on the clique members. On hard problems it may perform orders of magnitude better than `"ForcedColoring"` → None.
- `"ForcedColoring"` → `"LargestClique"` finds a largest clique, and forces colours on the clique members.
- `"ForcedColoring"` → None does not force any colours. It is usually the fastest choice for easy problems.

The default setting for `"ForcedColoring"` is `"MaxDegreeClique"`.

The Moser spindle is not 3-colourable, so no solution is returned.

In[1067]:=
```
moser = GraphData["MoserSpindle"];
```

In[1068]:=
```
IGKVertexColoring[moser, 3]
```

Out[1068]=
```
{}
```

Find a 4-colouring of the Moser spindle ...

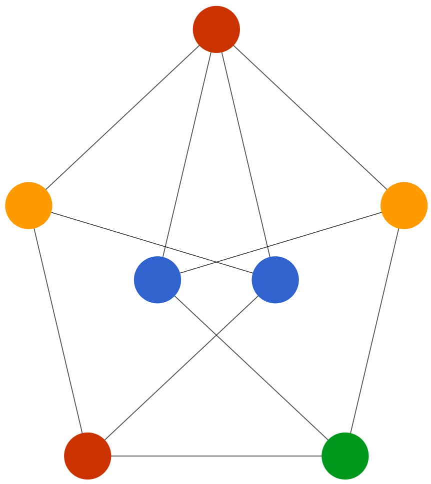
In[1069]:=
```
IGKVertexColoring[moser, 4]
```

Out[1069]=
```
{{4, 1, 3, 1, 3, 2, 2}}
```

... and visualize it.

In[1070]:=
```
Graph[moser, GraphStyle → "BasicBlack", VertexSize → Large] //
 IGVertexMap[ColorData[112], VertexStyle → (First@IGKVertexColoring[#, 4] &)]
```

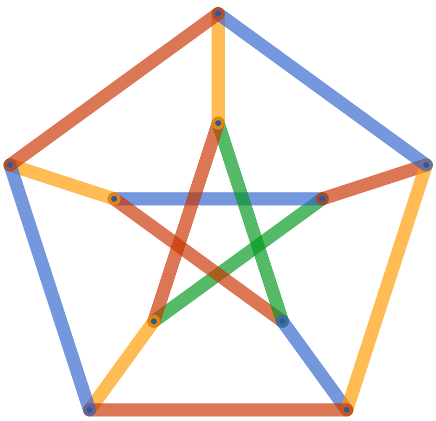
Out[1070]=



Find a 4-edge-colouring of the Petersen graph.

In[1071]:=
```
PetersenGraph[GraphStyle → "ThickEdge", EdgeStyle → Opacity[2 / 3]] //
 IGEdgeMap[ColorData[112], EdgeStyle → (First@IGKEdgeColoring[#, 4] &)]
```

Out[1071]=



The following examples illustrate the use of the "`ForcedColoring`" option. The 6th order Mycielski graph has chromatic number 6. A 6-colouring is easily found even with `"ForcedColoring" → None`.

In[1072]:=

```
g = GraphData[{"Mycielski", 6}];
```

In[1073]:=

```
IGKVertexColoring[g, 6, "ForcedColoring" → None] // Timing
```

Out[1073]=

```
{0.008607, {{6, 4, 5, 5, 4, 4, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3,
   3, 3, 3, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1}}}
```

However, showing that the graph is not 5-colourable takes considerably longer.

In[1074]:=

```
TimeConstrained[IGKVertexColoring[g, 5, "ForcedColoring" → None], 5]
```

Out[1074]=

```
$Aborted
```

Forcing colours in the appropriate way reduces the computation time significantly.

In[1075]:=

```
IGKVertexColoring[g, 5, "ForcedColoring" → "MaxDegreeClique"] // Timing
```

Out[1075]=

```
{0.31951, {}}
```

## Minimum colouring

In[1076]:=

```
? IGMinimumVertexColoring
```

IGMinimumVertexColoring[graph] gives a minimum vertex colouring of graph.

In[1077]:=

```
? IGMinimumEdgeColoring
```

IGMinimumEdgeColoring[graph] gives a minimum edge colouring of graph.

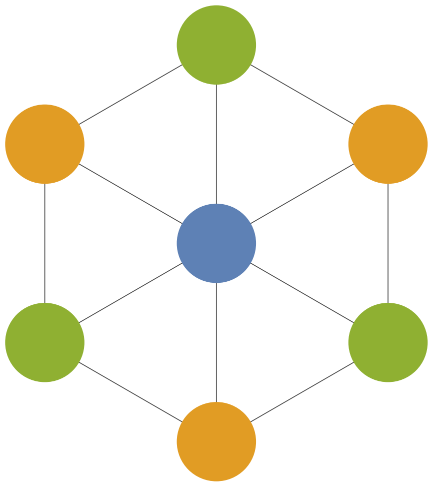
`IGMinimumVertexColoring` and `IGMinimumEdgeColoring` find minimum colourings of graphs, i.e. they find a colouring with the fewest possible number of colours. The current implementation tries successively larger *k*-colourings until it is successful.

`IGMinimumVertexColoring` and `IGMinimumEdgeColoring` can use the same `"ForcedColoring"` option values as `IGKVertexColoring` and `IGKEdgeColoring`.

In[1078]:=

```
WheelGraph[7, GraphStyle → "BasicBlack", VertexSize → Large] //
 IGVertexMap[ColorData[97], VertexStyle → IGMinimumVertexColoring]
```

Out[1078]=



Find a colouring of a large graph.

In[1079]:=

```
IGMinimumVertexColoring@RandomGraph[{100, 400}]
```

Out[1079]=

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

Implement a multipartite graph layout: vertex colouring is equivalent to partitioning the vertices of the graph into groups such that all connections run between different groups, and never within the same group. The colours can be thought of as the indices of groups. `IGMembershipToPartitions` can be used to convert from a group-index (i.e. membership) representation to a partition representation.

In[1080]:=

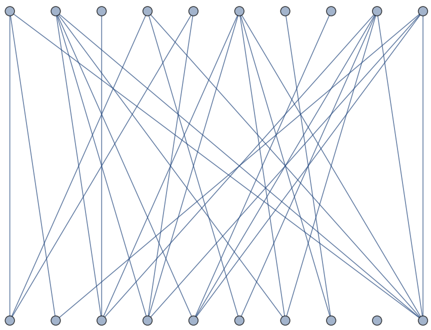
```
multipartiteLayout[g_?GraphQ, separation : _?NumericQ : 1.5, opt : OptionsPattern[]] :=
  Module[{n, partitions, partitionSizes, vertexCoordinates},
   partitions = IGMembershipToPartitions[g]@IGMinimumVertexColoring[g];
   partitionSizes = Length /@ partitions;
   n = Length[partitions];
   vertexCoordinates = With[{hl = N@Sin[Pi / n], ir = separation If[n == 2, 1 / 2, N@Cos[Pi / n]]},
     Catenate@Table[
       RotationTransform[2 Pi / n k][{#, ir} & /@ Subdivide[-hl, hl, partitionSizes[[k]] - 1]],
       {k, 1, n}
      ]
    ];
   IGReorderVertices[Catenate[partitions], g, VertexCoordinates → vertexCoordinates, opt]
  ];
```

Lay out a bipartite graph.

In[1081]:=

```
g = IGBipartiteGameGNM[10, 10, 30];
multipartiteLayout[g]
```

Out[1082]=

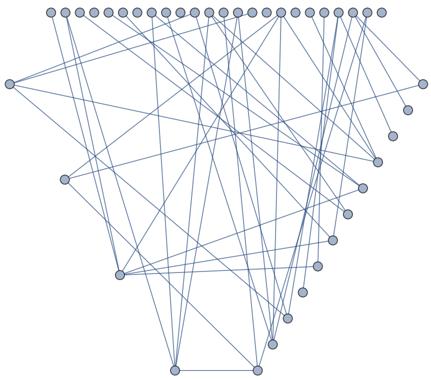


Lay out multipartite graphs.

In[1083]:=

```
g = RandomGraph[{40, 40}];
multipartiteLayout[g]
```

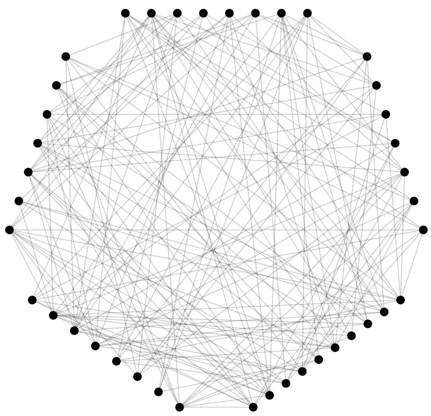
Out[1084]=



In[1085]:=

```
g = RandomGraph[{40, 160}];
multipartiteLayout[g, GraphStyle → "BasicBlack", EdgeStyle → Opacity[0.2]]
```

Out[1086]=

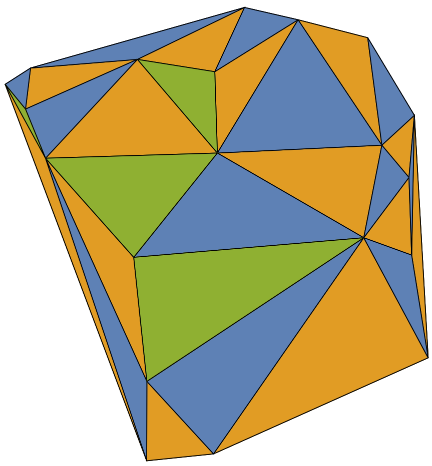


Compute a minimum colouring of a triangulation. It can be shown, e.g. based on Brooks's theorem, that any triangulation of a polygon is 3-colourable.

In[1087]:=

```
mesh = DelaunayMesh[RandomReal[1, {20, 2}], MeshCellStyle → {1 → Black}];
col = IGMinimumVertexColoring@IGMeshCellAdjacencyGraph[mesh, 2];
SetProperty[{mesh, {2, All}}, MeshCellStyle → ColorData[97] /@ col]
```

Out[1089]=

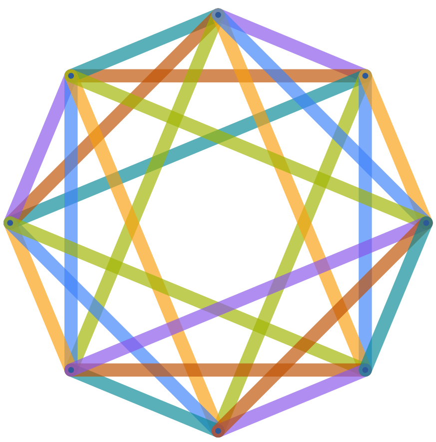


Find a minimum edge colouring of a graph.

In[1090]:=

```
Graph[
  GraphData["SixteenCellGraph"],
  GraphStyle → "ThickEdge", EdgeStyle → Opacity[2 / 3]
 ] //
 IGEdgeMap[
  ColorData[104],
  EdgeStyle → IGMinimumEdgeColoring
 ]
```

Out[1090]=



## Chromatic number

In[1091]:=

```
? IGChromaticNumber
```

IGChromaticNumber[graph] gives the chromatic number of graph.

In[1092]:=

```
? IGChromaticIndex
```

IGChromaticIndex[graph] gives the chromatic index of graph.

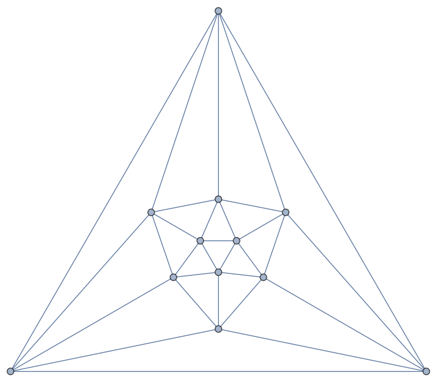
The chromatic number of a graph is the smallest number of colours needed to colour its vertices. The chromatic index, or edge chromatic number, is the smallest number of colours needed to colour its edges.

Find the chromatic number and chromatic index of a graph.

In[1093]:=

```
g = GraphData["IcosahedralGraph"]
```

Out[1093]=



In[1094]:=

```
{IGChromaticNumber[g], IGChromaticIndex[g]}
```

Out[1094]=

```
{4, 5}
```

The implementation of `IGChromaticNumber` and `IGChromaticIndex` is effectively the following:

In[1095]:=

```
{Max@IGMinimumVertexColoring[g], Max@IGMinimumEdgeColoring[g]}
```

Out[1095]=

```
{4, 5}
```

## Perfect graphs

In[1096]:=

```
? IGPerfectQ
```

IGPerfectQ[graph] tests if graph is perfect. The chromatic number and the clique number are the same in every induced subgraph of a perfect graph.

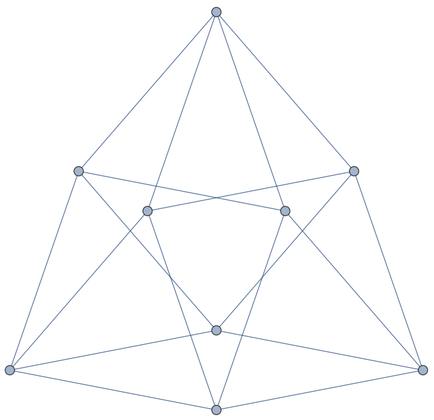
`IGPerfectQ` tests if a graph is perfect. The clique number and the chromatic number is the same for every induced subgraph of a perfect graph.

The current implementation of `IGPerfectQ` uses the strong perfect graph theorem: it checks that neither the graph nor its complement have a graph hole of odd length.

In[1097]:=

```
g = GraphData[{"GeneralizedQuadrangle", {2, 1}}]
```

Out[1097]=



In[1098]:=

```
IGPerfectQ[g]
```

Out[1098]=

```
True
```

The clique number and the chromatic number is the same for every induced subgraph.

In[1099]:=

```
AllTrue[
 Subgraph[g, #] & /@ Subsets@VertexList[g],
 IGCliqueNumber[#] == IGChromaticNumber[#] &
]
```

Out[1099]=

```
True
```

## Utility functions

In[1100]:=

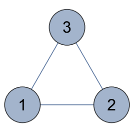
```
? IGVertexColoringQ
```

IGVertexColoringQ[graph, coloring] tests whether neighbouring vertices all have differing colours.

`IGVertexColoringQ` checks whether neighbouring vertices of a graph are assigned different colours.

The colours may be given as a list, with the same ordering as `VertexList[graph]`.

In[1101]:=

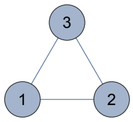
```
IGVertexColoringQ[
```


```
, {1, 2, 3}]
```

Out[1101]=

```
True
```

In[1102]:=

```
IGVertexColoringQ[
```


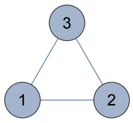
```
, {1, 2, 2}]
```

Out[1102]=

```
False
```

The colours may also be given as an association from vertices to colours.

In[1103]:=

```
IGVertexColoringQ[
```


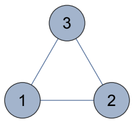
```
, <|1 → 3, 2 → 2, 3 → 1|>]
```

Out[1103]=

```
True
```

Any expression may be used for the colours, not only integers.

In[1104]:=

```
IGVertexColoringQ[
```


```
, <|1 → "a", 2 → "b", 3 → "c"|>]
```

Out[1104]=

```
True
```

# Processes on graphs

## Random walks

### IGRandomWalk

In[1105]:=

```
? IGRandomWalk
```

IGRandomWalk[graph, start, steps] takes a random walk of length steps on graph, starting at vertex 'start'. The list of traversed vertices is returned.

`IGRandomWalk[]` takes a random walk over a directed or undirected graph. If the graph is weighted, the next edge to traverse is selected with probability proportional to its weight.

The available options are:

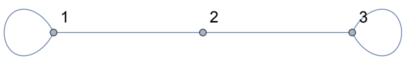
- `EdgeWeight` can be used to override the existing weights of the graph. `EdgeWeight → None` will ignore any existing weights.

Traversing self-loops in different directions is considered as distinct probabilities in an undirected graph. Thus vertices 1 and 3 are visited more often in the below graphs than vertex 2:

In[1106]:=

```
g = Graph[{1 ⟷ 1, 1 ⟷ 2, 2 ⟷ 3, 3 ⟷ 3}, VertexLabels → "Name"]
```

Out[1106]=



In[1107]:=

```
IGRandomWalk[g, 1, 100 000] // Counts // KeySort
```

Out[1107]=

```
<|1 → 37 453, 2 → 25 082, 3 → 37 465|>
```

This is consistent with their degrees:

In[1108]:=

```
VertexDegree[g]
```

Out[1108]=

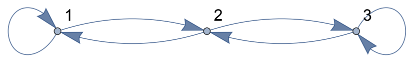
```
{3, 2, 3}
```

Convert the graph to a directed version to traverse self-loops only in one direction.

In[1109]:=

```
dg = DirectedGraph[g]
```

Out[1109]=




In[1110]:=

```
{VertexOutDegree[dg], VertexInDegree[dg]}
```

Out[1110]=

```
{{2, 2, 2}, {2, 2, 2}}
```

In[1111]:=

```
IGRandomWalk[dg, 1, 100 000] // Counts // KeySort
```

Out[1111]=

```
<|1 → 33 345, 2 → 33 317, 3 → 33 338|>
```

If the walker gets stuck, a list shorter than `steps` will be returned. This may happen in a non-connected directed graph, or in a single-vertex graph component.

In[1112]:=

```
IGRandomWalk[IGEmptyGraph[1], 1, 10]
```

Out[1112]=

```
{1}
```

In[1113]:=

```
IGRandomWalk[Graph[{1 → 2}], 1, 10]
```

Out[1113]=

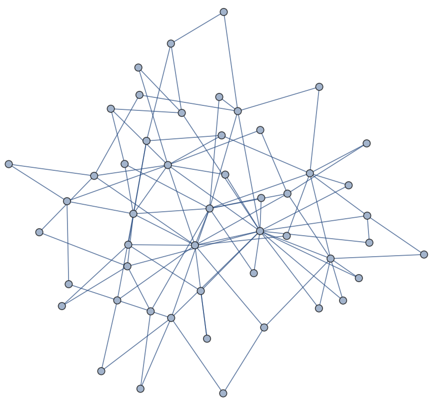
```
{1, 2}
```

How much time does a random walker spend on each node of a network?

In[1114]:=

```
g = IGBarabasiAlbertGame[50, 2, DirectedEdges → False]
```

Out[1114]=



In[1115]:=

```
counts = Counts@IGRandomWalk[g, First@VertexList[g], 10 000] /@ VertexList[g]
```

Out[1115]=

```
{591, 550, 411, 906, 545, 198, 261, 411, 242, 420, 151, 247, 215, 171, 280,
 113, 88, 216, 277, 133, 115, 158, 103, 99, 143, 93, 100, 123, 254, 188, 155, 175,
 89, 113, 123, 96, 100, 87, 141, 81, 88, 93, 98, 111, 106, 115, 101, 96, 125, 105}
```

The exact answer can be computed as the leading eigenvector of the process's stochastic matrix:

In[1116]:=

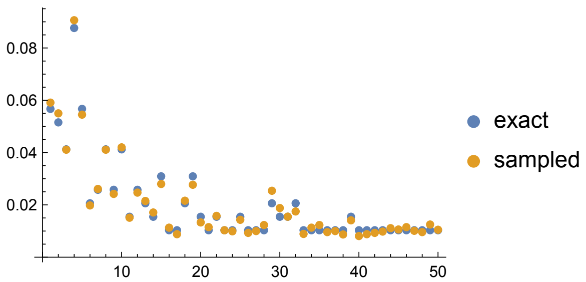
```
sm = Transpose[AdjacencyMatrix[g] / VertexDegree[g]];
{{val}, {vec}} = Eigensystem[N[sm], 1, Method → {"Arnoldi", "Criteria" → "RealPart"}];
```

Compare the exact answer with a finite sample:

In[1118]:=

```
ListPlot[{vec / Total[vec], counts / Total[counts]}, PlotRange → All,
 PlotLegends → {"exact", "sampled"}, PlotStyle → PointSize[0.02]]
```

Out[1118]=

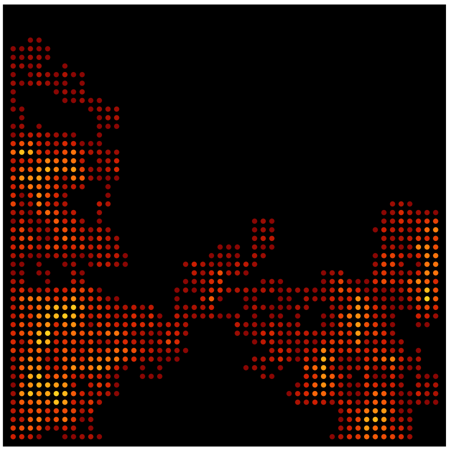


Random walk on a square grid.

In[1119]:=

```
grid = IGSquareLattice[{50, 50}];
counts = Counts@IGRandomWalk[grid, 1, 5000];
Graph[grid,
 VertexStyle →
  Prepend[
   Normal[ColorData["SolarColors"] /@ Normalize[counts, Max]],
   Black (* colour of unvisited nodes, i.e. default colour *)
  ],
 EdgeShapeFunction → None,
 Background → Black
]
```

Out[1121]=



The fraction of nodes reached after *n* steps on a grid and a comparable random regular graph.

In[1122]:=

```
nodesReached[graph_] :=
 Length@Union@IGRandomWalk[graph, 1, VertexCount[graph]] / VertexCount[graph]
```

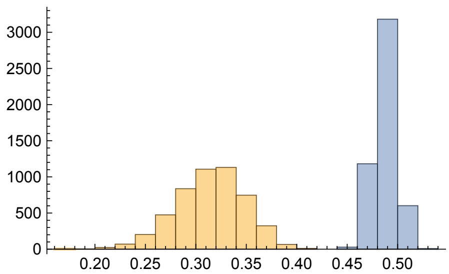
In[1123]:=

```
grid = IGSquareLattice[{50, 50}, "Periodic" → True];
regular = IGKRegularGame[50^2, 4];
```

In[1125]:=

```
Table[
   {nodesReached[grid], nodesReached[regular]},
   {5000}
  ] // Transpose // Histogram
```

Out[1125]=



Generate random spanning trees using loop erased random walks.

In[1126]:=

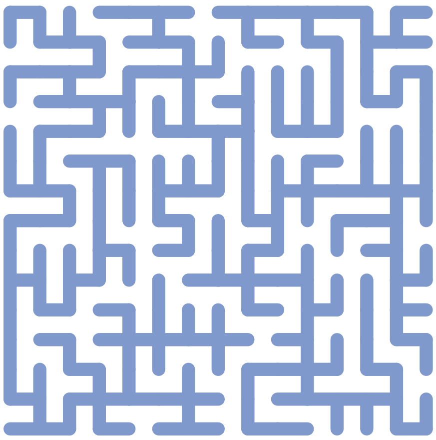
```
randomSpanningTree[graph_?GraphQ] :=
 Module[{visited = <||>, i = 2, k = 1, batchSize = 2 VertexCount[graph], walk},
  walk = IGRandomWalk[graph, RandomChoice@VertexList[graph], batchSize];
  visited[walk[[1]]] = True;
  While[k < VertexCount[graph],
      (* register a traversed edge only when it leads to a yet unvisited vertex *)
      If[! TrueQ[visited[walk[[i]]]],
       Sow[walk[[i - 1]] <-> walk[[i]]];
       visited[walk[[i]]] = True;
       k++
      ];
      i++;
      (* if the walk has not yet visited all vertices, keep walking *)
      If[i > Length[walk],
       walk = Join[walk, Rest@IGRandomWalk[graph, Last[walk], batchSize]]
      ];
     ] // Reap // Last // First
 ]
```

By taking random spanning trees of spatially embedded planar graphs, we can generate mazes.

In[1127]:=

```
graph = IGSquareLattice[{15, 15}];
Graph[VertexList[graph], randomSpanningTree[graph],
 VertexCoordinates -> GraphEmbedding[graph],
 GraphStyle -> "ThickEdge", VertexShapeFunction -> None, EdgeStyle -> ■
]
```

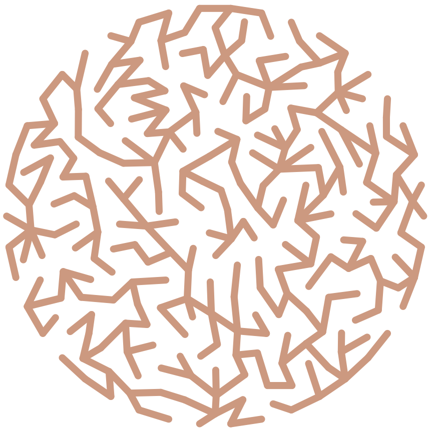
Out[1128]=



In[1129]:=

```
graph = IGMeshGraph@DiscretizeRegion@Disk[];
Graph[VertexList[graph], randomSpanningTree[graph],
 VertexCoordinates → GraphEmbedding[graph],
 GraphStyle → "ThickEdge", VertexShapeFunction → None,
 EdgeStyle → Directive[■, AbsoluteThickness[4]]
]
```

Out[1130]=



Take a sample of a large graph using a random walk. The following graph is too large to easily visualize, but visualizing a random-walk-based sample immediately shows signs of a community structure.

In[1131]:=

```
g = ExampleData[{"NetworkGraph", "AstrophysicsCollaborations"}];
{VertexCount[g], VertexCount@IGGiantComponent[g]}
```

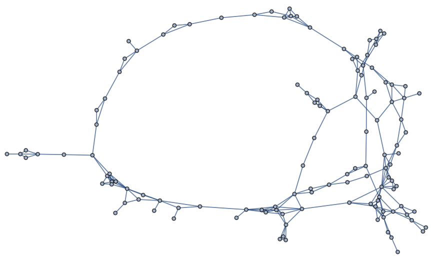
Out[1132]=

```
{16 706, 14 845}
```

In[1133]:=

```
Subgraph[g, IGRandomWalk[g, RandomChoice@VertexList@IGGiantComponent[g], 200]]
```

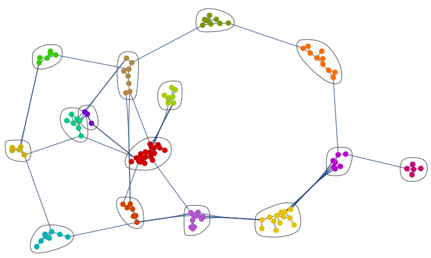
Out[1133]=



In[1134]:=

```
CommunityGraphPlot[%]
```

Out[1134]=



## IGRandomEdgeWalk and IGRandomEdgeIndexWalk

In[1135]:=

```
? IGRandomEdgeWalk
```

IGRandomEdgeWalk[graph, start, steps] takes a random walk of length steps on graph, starting at vertex 'start'. The list of traversed edges is returned.

In[1136]:=

```
? IGRandomEdgeIndexWalk
```

IGRandomEdgeIndexWalk[graph, start, steps] takes a random walk of length steps on graph, starting at vertex 'start'. The list of indices for traversed edges is returned.

`IGRandomEdgeWalk` takes a random walk on a graph and returns the list of traversed edges. If the graph is weighted, the next edge to traverse is selected with probability proportional to its weight.

The available options are:

- `EdgeWeight` can be used to override the existing weights of the graph. `EdgeWeight → None` will ignore any existing weights.

Take a random walk on a De Bruijn graph, and retrieve the traversed edges.

In[1137]:=

```
g = IGDeBruijnGraph[3, 3];
IGRandomEdgeWalk[g, RandomChoice@VertexList[g], 20]
```

Out[1138]=

```
{13 → 11, 11 → 6, 6 → 16, 16 → 21, 21 → 9, 9 → 25, 25 → 21, 21 → 7, 7 → 20,
 20 → 6, 6 → 16, 16 → 19, 19 → 1, 1 → 3, 3 → 7, 7 → 19, 19 → 2, 2 → 4, 4 → 11, 11 → 4}
```

`IGRandomEdgeIndexWalk` returns the list of indices of the traversed edges instead. This makes it useful for working with multigraphs, as it allows distinguishing between parallel edges.

As an example application, let us consider the following set of affine transformations:

In[1139]:=

```
scale12 = ScalingTransform[{1 / 2, 1 / 2}];
a11 = TranslationTransform[{1 / 4, √3 / 4}] @* scale12;
a21 = RotationTransform[Pi / 3] @* scale12;
b21 = TranslationTransform[{3 / 4, √3 / 4}] @* RotationTransform[-Pi / 3] @* scale12;
a12 = TranslationTransform[{1 / 2, 0}] @* ScalingTransform[{1 / 2, -1 / 2}];
a22 = scale12;
```

In[1145]:=

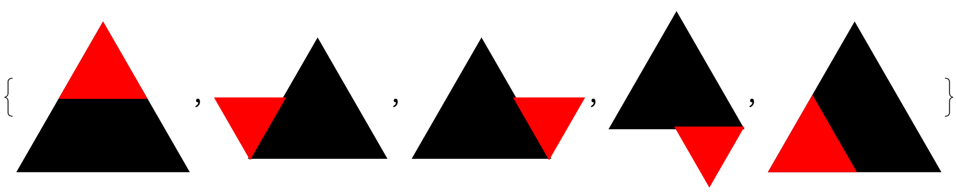
```
trafos = {a11, a21, b21, a12, a22};
```

Let us visualize them by showing an initial (black) triangle and its (red) transformation.

In[1146]:=

```
tri = Triangle[{{0, 0}, {1, 0}, {1 / 2, √3 / 2}}];
Graphics[{tri, Red, GeometricTransformation[tri, #]}, ImageSize → Tiny] & /@ trafos
```

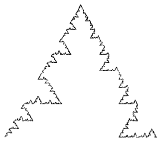
Out[1147]=



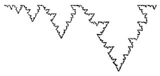
These transformations describe the mutual self-similarity structure of two fractal curves, according to the following directed graph. Each edge of the graph corresponds to a transformation.

In[1148]:=

```
graph = Graph[{1 → 1, 2 → 1, 2 → 1, 1 → 2, 2 → 2},
  VertexLabels → Placed["Name", Center],
  VertexShape → {1 →
```

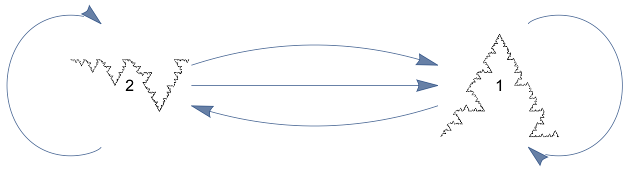


```
, 2 →
```



```
}, PerformanceGoal → "Quality",
  VertexSize → 1/3, ImageSize → 400
 ]
```

Out[1148]=




Let us compute a random walk on this graph, and iteratively apply transformations to the point {0, 0} according to the traversed edges.

In[1149]:=

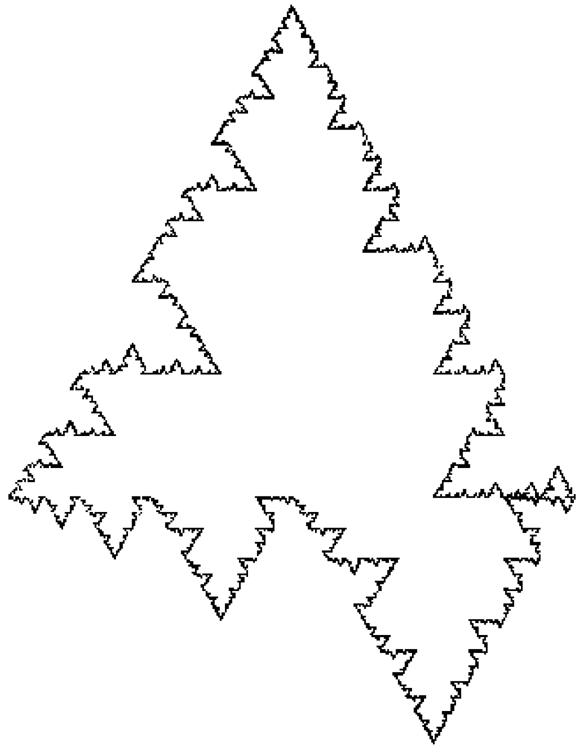
```
walk = IGRandomEdgeIndexWalk[graph, 1, 20 000];
pts = Rest@FoldList[#2[#1] &, {0., 0.}, trafos[[walk]]];
```

The resulting list of points will approximate the union of the two fractal curves.

In[1151]:=

```
Image@Graphics[{AbsolutePointSize[1], Point[pts]}]
```

Out[1151]=



The two curves can be separated by filtering points according to which graph vertex the corresponding directed edge targets. For example, if the point was generated by a transformation corresponding to 1 → 2, it will belong to curve 2.

In[1152]:=

```
targets = Last /@ EdgeList[graph]
```

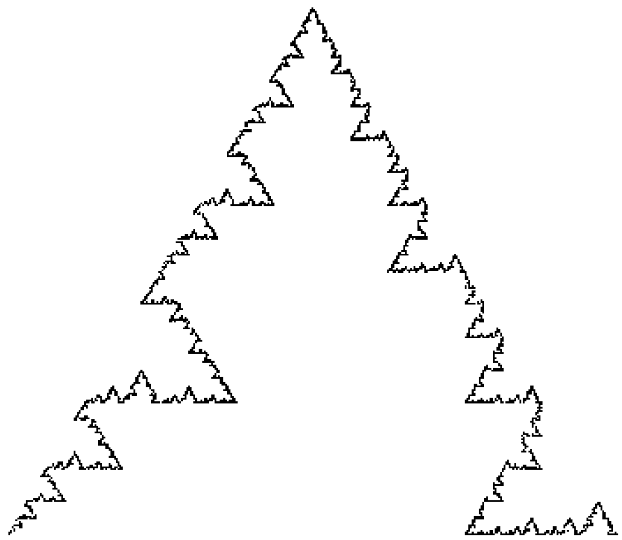
Out[1152]=

```
{1, 1, 1, 2, 2}
```

In[1153]:=

```
Image@Graphics[{AbsolutePointSize[1], Point@Pick[pts, targets[[walk]], 1]}]
```

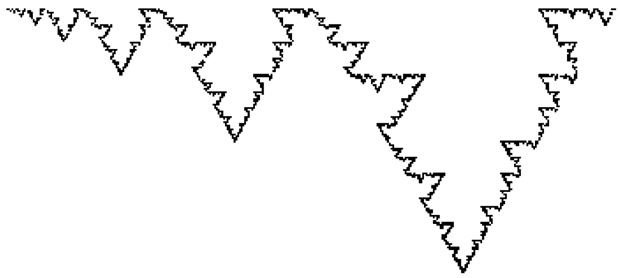
Out[1153]=



In[1154]:=

```
Image@Graphics[{AbsolutePointSize[1], Point@Pick[pts, targets[[walk]], 2]}]
```

Out[1154]=



The technique described here is taken from “Generating self-affine tiles and their boundaries” by Mark McClure.

## Epidemic models

### IGSIRProcess

In[1155]:=

```
? IGSIRProcess
```

IGSIRProcess[graph, {β, γ}] runs a stochastic epidemic SIR model on graph with infection rate β and recovery rate γ, and returns a time series of {S, I, R} values.

IGSIRProcess[graph, {β, γ}, n] performs n SIR model runs.

`IGSIRProcess` simulates a stochastic version of the well known SIR model of disease spreading. In this model, each node of the network may be in one of three states: susceptible, infected or recovered, denoted by *S*, *I* and *R*, respectively. A susceptible node with *k* infected neighbours becomes infected with rate $k\beta$, while an infected node recovers with rate $\gamma$. At the start of the simulation, a random node is chosen to be infected. The simulation runs until no more infected nodes are left.

When performing a single simulation, `IGSIRProcess` returns a TimeSeries expression of `{s, i, r}` values. When multiple runs are requested, the resulting time series are combined into a TemporalData expression.

In[1156]:=

```
g = IGWattsStrogatzGame[100, 0.05];
```

Perform a single SIR simulation:

In[1157]:=

```
ts = IGSIRProcess[g, {5, 1}]
```

Out[1157]=

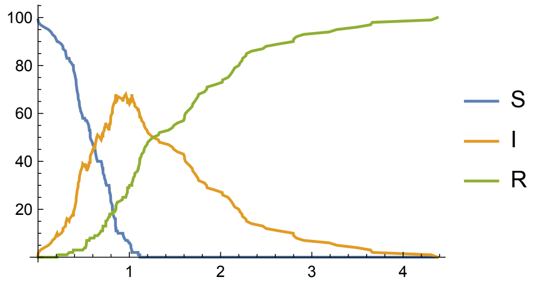
```
TimeSeries[ Time: 0. to 4.37  Data points: 200 ]
```

Plot the results with a legend:

In[1158]:=

```
ListLinePlot[ts, PlotLegends → ts["ComponentNames"]]
```

Out[1158]=

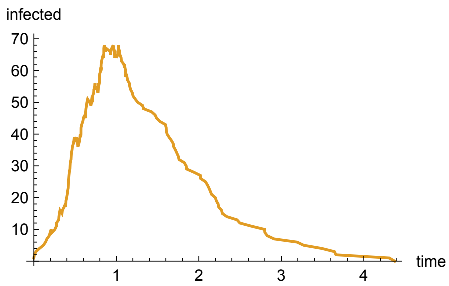



Plot only the number of infected nodes:

In[1159]:=

```
(* In Mathematica 12.0 and later,
   ts["PathComponent", "I"] can also be used. *)
ListLinePlot[ts["PathComponent", 2],
 AxesLabel → {"time", "infected"}, PlotStyle → ColorData[97][2]]
```

Out[1159]=




Find the number of susceptible, infected and recovered nodes at a specific time point:

In[1160]:=

```
ts[1.0]
```

Out[1160]=

```
{6., 65., 29.}
```

The ResamplingMethod of the TimeSeries object is set to 0th order interpolation, therefore the last value is used beyond the last available time point.

In[1161]:=

```
ts[10]
```

InterpolatingFunction: Input value {–10} lies outside the range of data in the interpolating function. Extrapolation will be used.

Out[1161]=

```
{0., 0., 100.}
```

Perform 100 simulations simultaneously:

In[1162]:=

```
td = IGSIRProcess[g, {5, 1}, 100]
```

Out[1162]=

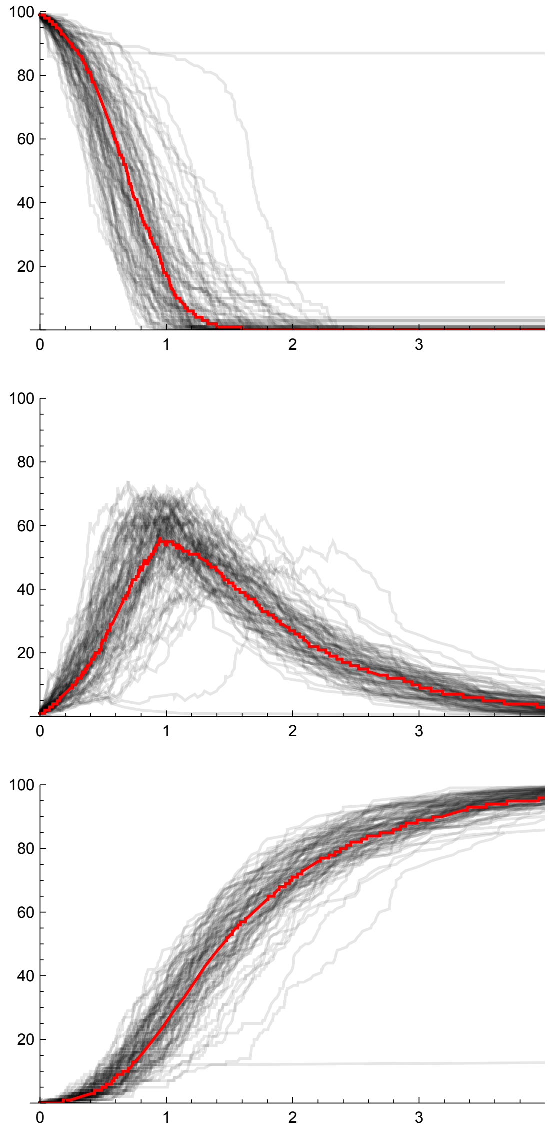
```
TemporalData[ Time: 0. to 9.26  Data points: 17 970  Paths: 100 ]
```

Plot the median number of susceptible, infected and recovered nodes:

In[1163]:=

```
Show[
    ListLinePlot[#, PlotStyle → GrayLevel[0, 0.1], PlotRange → {0, VertexCount[g]}],
    Quiet@Plot[Median[#[t]], {t, 0, 4}, PlotStyle → Red]
   ] & /@ td["PathComponents"] // GraphicsColumn
```

Out[1163]=



The sum of the three components, $S + I + R$, always equals the total number of graph nodes.

In[1164]:=

```
First@Normal@Total[td["PathComponents"]] // Short
```

Out[1164]//Short=

```
{{0., 100.}, {0.00200952, 100.}, ≪197≫, {5.6842, 100.}}
```

In the next example, we compare epidemic spreading on a periodic grid, i.e. a network that only has spatially local connections, with a rewired version of the same network which also includes long range links. We rewire 5% of links while ensuring that the graph stays connected.

In[1165]:=

```
g1 = IGSquareLattice[{30, 30}, "Periodic" → True];
g2 = IGTryUntil[IGConnectedQ][IGRewireEdges[g1, 0.05]];
```

Generate 1000 simulations for each network.

In[1167]:=

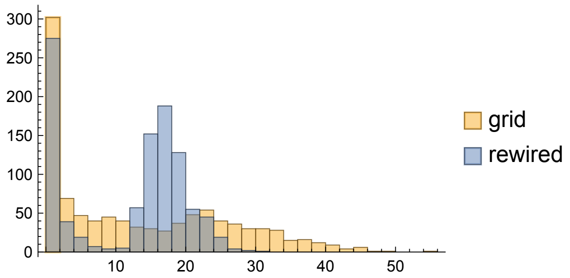
```
r1 = IGSIRProcess[g1, {1, 1}, 1000];
r2 = IGSIRProcess[g2, {1, 1}, 1000];
```

Plot the histogram of the total duration of the epidemic.

In[1169]:=

```
Histogram[{r1["LastTimes"], r2["LastTimes"]}, ChartLegends → {"grid", "rewired"}]
```

Out[1169]=

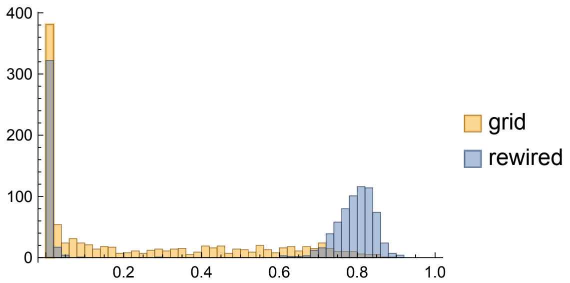



Plot the fraction of recovered nodes at the end of the epidemic.

In[1170]:=

```
tmax = Max[r1["MaximumTime"], r2["MaximumTime"]];
Histogram[
 {r1["PathComponent", 3]["SliceData", tmax] / VertexCount[g1],
   r2["PathComponent", 3]["SliceData", tmax] / VertexCount[g2]} // Quiet,
 {0, 1, 0.02},
 ChartLegends → {"grid", "rewired"}
]
```

Out[1171]=




## Planar graphs

A graph is said to be *planar* if it can be drawn in the plane without edge crossings.

A useful concept when working with planar graphs is their combinatorial embedding. A combinatorial embedding of a graph is a counter-clockwise ordering of the incident edges around each vertex. IGraph/M represents combinatorial embeddings as associations from vertices to an ordering of their neighbours. Currently, only embeddings of simple graphs are supported.

Some of the planar graph functionality makes use of the LEMON Graph Library.

## IGPlanarQ

In[1172]:=

```
? IGPlanarQ
```

IGPlanarQ[graph] tests if graph is planar.
IGPlanarQ[embedding] tests if a combinatorial embedding is planar.

`IGPlanarQ[graph]` checks if a graph is planar using the Boyer–Myrvold algorithm.

In[1173]:=

```
IGPlanarQ@GraphData[{"Apollonian", 6}]
```

Out[1173]=

```
True
```

In[1174]:=

```
IGPlanarQ@CompleteGraph[5]
```

Out[1174]=

```
False
```

`IGPlanarQ[embedding]` checks if a combinatorial embedding is planar. The following are both embeddings of the $K_4$ complete graph. However, only the first one is planar.

In[1175]:=

```
emb1 = <|1 → {2, 3, 4}, 2 → {1, 4, 3}, 3 → {2, 4, 1}, 4 → {3, 2, 1}|>;
emb2 = <|1 → {2, 4, 3}, 2 → {4, 3, 1}, 3 → {1, 2, 4}, 4 → {3, 1, 2}|>;
```

In[1177]:=

```
IGPlanarQ /@ {emb1, emb2}
```

Out[1177]=

```
{True, False}
```

The second embedding generates only 2 faces instead of 4, which can be embedded on a torus, but not in the plane (or on a sphere).

In[1178]:=

```
Length /@ IGFaces /@ {emb1, emb2}
```

Out[1178]=

```
{4, 2}
```

Unlike the built-in `PlanarGraphQ`, `IGPlanarQ` considers the null graph to be planar.

In[1179]:=

```
{IGPlanarQ@IGEmptyGraph[], PlanarGraphQ@IGEmptyGraph[]}
```

Out[1179]=

```
{True, True}
```

## IGMaximalPlanarQ

In[1180]:=

```
? IGMaximalPlanarQ
```

IGMaximalPlanarQ[graph] tests if graph is maximal planar.

A simple graph is *maximal planar* if no new edges can be added to it without breaking planarity. Maximal planar graphs are sometimes called *triangulated* graphs or *triangulations*.

The 3-cycle is maximal planar.

In[1181]:=
```
IGMaximalPlanarQ[CycleGraph[3]]
```
Out[1181]=
```
True
```

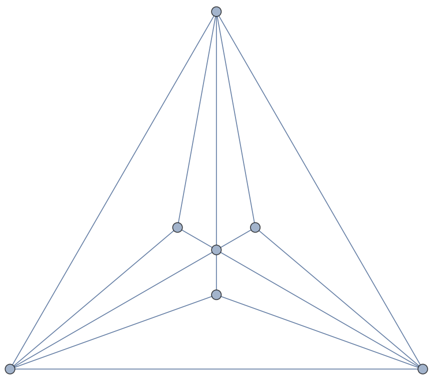
The 4-cycle is not because a chord can be added to it without breaking planarity.

In[1182]:=
```
IGMaximalPlanarQ[CycleGraph[4]]
```
Out[1182]=
```
False
```

In[1183]:=
```
IGPlanarQ[EdgeAdd[CycleGraph[4], 1 <-> 3]]
```
Out[1183]=
```
True
```

Apollonian graphs are maximal planar.

In[1184]:=
```
g = GraphData[{"Apollonian", 2}]
```
Out[1184]=



In[1185]:=
```
IGMaximalPlanarQ[g]
```
Out[1185]=
```
True
```

All faces of a maximal planar graph are triangles.

In[1186]:=
```
Length /@ IGFaces[g]
```
Out[1186]=
```
{3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
```

Therefore the edge count $E$ and the face count $F$ of a maximal planar graph on more than 2 vertices satisfy $2\,E = 3\,F$. Each edge is incident to two faces and each face is incident to three edges.

In[1187]:=
```
{2 EdgeCount[g], 3 Length@IGFaces[g]}
```
Out[1187]=
```
{30, 30}
```

## IGOuterplanarQ

In[1188]:=

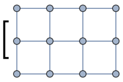
```
? IGOuterplanarQ
```

IGOuterplanarQ[graph] tests if graph is outerplanar.
IGOuterplanarQ[embedding] tests if a combinatorial embedding is outerplanar.

`IGOuterplanarQ[graph]` checks if a graph is outerplanar, i.e. if it can be drawn in the plane without edge crossings and with all vertices being on the outer face.

Outerplanar graphs are also called circular planar. They can be drawn without edge crossings and all vertices on a circle. See the documentation of `IGOuterplanarEmbedding` for an example.

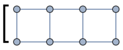
In[1189]:=

`IGOuterplanarQ[`  `]`

Out[1189]=

```
False
```

In[1190]:=

`IGOuterplanarQ[`  `]`

Out[1190]=

```
True
```

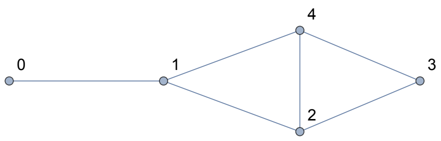
`IGOuterplanarQ[embedding]` checks if a combinatorial embedding is outerplanar. Not all planar embeddings of an outerplanar graph are also outerplanar embeddings.

Consider the following outerplanar graph …

In[1191]:=

```
g = IGShorthand["0-1-2-3-4-2,1-4"]
```

Out[1191]=




In[1192]:=

```
IGOuterplanarQ[g]
```

Out[1192]=

```
True
```

… and two of its embeddings:

In[1193]:=

```
emb1 = <|0 → {1}, 1 → {2, 0, 4}, 2 → {1, 3, 4}, 3 → {2, 4}, 4 → {3, 1, 2}|>;
emb2 = <|0 → {1}, 1 → {0, 2, 4}, 2 → {1, 3, 4}, 3 → {2, 4}, 4 → {3, 1, 2}|>;
```

They are both planar, but only the second one is outerplanar.

In[1195]:=

```
IGPlanarQ /@ {emb1, emb2}
```

Out[1195]=

```
{True, True}
```

In[1196]:=

```
IGOuterplanarQ /@ {emb1, emb2}
```

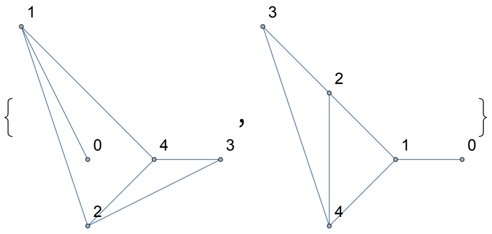
Out[1196]=

```
{False, True}
```

In[1197]:=

```
Graph[g, VertexCoordinates → IGEmbeddingToCoordinates[#]] & /@ {emb1, emb2}
```

Out[1197]=




## IGKuratowskiEdges

In[1198]:=

```
? IGKuratowskiEdges
```

IGKuratowskiEdges[graph] gives the edges belonging to a Kuratowski subgraph.

`IGKuratowskiEdges` finds a Kuratowski subgraph of a non-planar graph. The subgraph is returned as a set of edges. If the graph is planar, `{}` is returned.

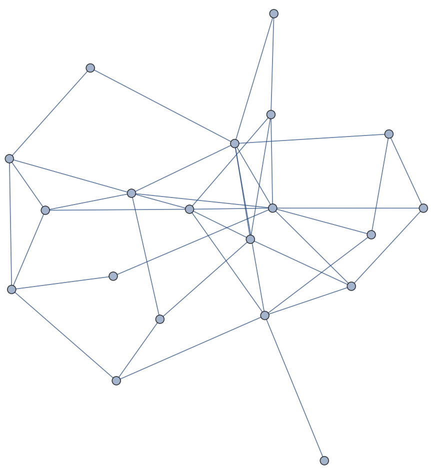
According to Kuratowski's theorem, any non-planar graph contains a subgraph homeomorphic to the $K_5$ complete graph or the $K_{3,3}$ complete bipartite graph. This is called a Kuratowski subgraph.

Generate a random graph, which is non-planar with high probability.

In[1199]:=

```
g = RandomGraph[{20, 40}]
```

Out[1199]=



In[1200]:=

```
IGPlanarQ[g]
```

Out[1200]=

```
False
```

Compute a set of edges belonging to a Kuratowski subgraph.

In[1201]:=

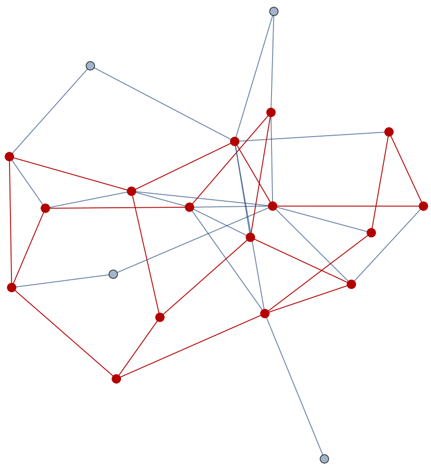
```
kur = IGKuratowskiEdges[g]
```

Out[1201]=

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

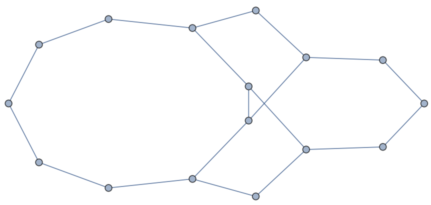
Highlight the Kuratowski subgraph.

In[1202]:=
```
HighlightGraph[g, Graph[kur]]
```
Out[1202]=



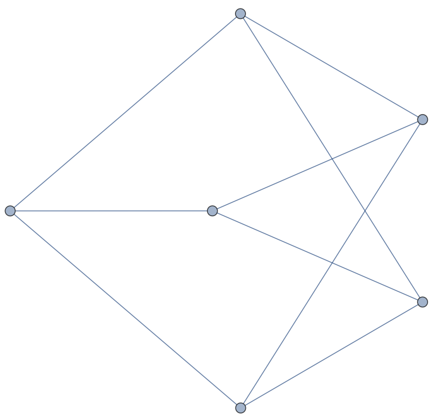
Display the Kuratowski subgraph on its own.

In[1203]:=
```
Graph[kur]
```
Out[1203]=



By smoothening the Kuratowski subgraph, we obtain either $K_5$ or $K_{3,3}$.

In[1204]:=
```
IGSmoothen@Graph[kur]
```
Out[1204]=



In[1205]:=
```
IGHomeomorphicQ[Graph[kur], #] & /@ {CompleteGraph[5], CompleteGraph[{3, 3}]}
```
Out[1205]=
```
{False, True}
```

For planar graphs, { } is returned.

In[1206]:=

```
IGKuratowskiEdges@CycleGraph[5]
```

Out[1206]=

```
{}
```

## IGFaces

In[1207]:=

```
? IGFaces
```

IGFaces[graph] gives the faces of a planar graph.
IGFaces[embedding] gives the faces that correspond to a combinatorial embedding.

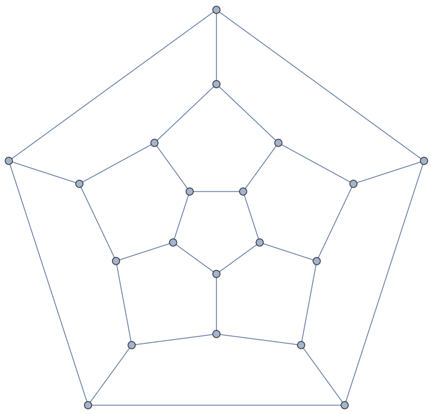
`IGFaces` returns the faces of a planar graph, or the faces corresponding to a specific (not necessarily planar) embedding. The faces are represented by a counter-clockwise ordering of vertices. The current implementation ignores self-loops and multi-edges.

The faces of a planar graph are unique if the graph is 3-vertex-connected. This can be checked using `KVertexConnectedGraphQ`.

In[1208]:=

```
g = GraphData["DodecahedralGraph"]
```

Out[1208]=



In[1209]:=

```
IGFaces[g]
```

Out[1209]=

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

In[1210]:=

```
KVertexConnectedGraphQ[g, 3]
```

Out[1210]=

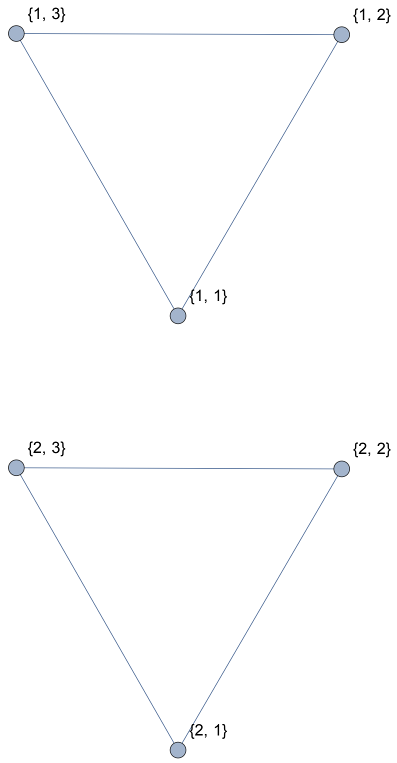
```
True
```

If the graph is not connected and has $C$ connected components, then $C - 1$ faces will be redundant.

In[1211]:=

```
g = IGDisjointUnion[{CycleGraph[3], CycleGraph[3]}, VertexLabels → Automatic]
```

Out[1211]=




In[1212]:=

```
IGFaces[g]
```

Out[1212]=

```
{{{1, 1}, {1, 2}, {1, 3}}, {{1, 1}, {1, 3}, {1, 2}}, {{2, 1}, {2, 2}, {2, 3}}, {{2, 1}, {2, 3}, {2, 2}}}
```

In the above-drawn arrangement, the outer faces of the two triangles are the same face. However, one triangle could have been drawn inside of the other. Then the inner face of one would be the same as the outer face of the other. Thus the choice of faces to be eliminated as redundant is arbitrary, and is left up to the user.

`IGFaces` can also be used with a non-planar combinatorial embedding. The below embeddings both belong to the 4-vertex complete graph, however, only the first is planar.

In[1213]:=

```
emb1 = <|1 → {2, 3, 4}, 2 → {1, 4, 3}, 3 → {2, 4, 1}, 4 → {3, 2, 1}|>;
emb2 = <|1 → {2, 4, 3}, 2 → {4, 3, 1}, 3 → {1, 2, 4}, 4 → {3, 1, 2}|>;
```

In[1215]:=

```
IGFaces[emb1]
```

Out[1215]=

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

In[1216]:=

```
IGFaces[emb2]
```

Out[1216]=

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

Determine the genus $g$ of an embedding belonging to a connected graph based on its face count $F$, vertex count $V$, and edge count $E$, using the formula for the Euler characteristic $2g - 2 = \chi = V - E + F$.

In[1217]:=

```
genus[emb_?IGEmbeddingQ] := (2 + Total[Length /@ emb] / 2 - Length[emb] - Length@IGFaces[emb]) / 2
```

In[1218]:=

```
genus /@ {emb1, emb2}
```

Out[1218]=

```
{0, 1}
```

## IGDualGraph

In[1219]:=

```
? IGDualGraph
```

IGDualGraph[graph] gives the dual graph of a planar graph.
IGDualGraph[embedding] gives the dual graph corresponding to a specific embedding of a graph. The embedding does not need to be planar.

`IGDualGraph` returns a dual graph of a planar graph, or the dual corresponding to a specific embedding. The ordering of the dual graph's vertices is consistent with the result of `IGFaces`.

Limitations:

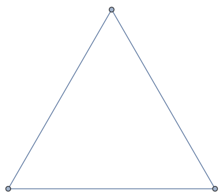
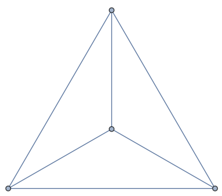
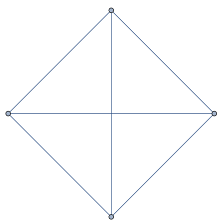
- Multi-edges and self-loops are currently ignored.
- The result is always a simple graph. No multi-edges or self-loops are generated

The dual of a simple 3-vertex-connected graph is simple and unique, thus such graphs are not affected by the above limitations.

In[1220]:=

```
TableForm[
 Table[{CompleteGraph[k], IGDualGraph@CompleteGraph[k]}, {k, 1, 4}],
 TableHeadings → {None, {"graph", "dual"}}
]
```

Out[1220]//TableForm=

| graph | dual |
|---|---|
|  | |
|  |  |
|  |  |
|  |  |

Currently, if the input is a graph, it must be planar.

In[1221]:=

```
IGDualGraph[CompleteGraph[5]]
```

IGraphM: planarEmbedding: The graph is not planar.

Out[1221]=

```
$Failed
```

If the input is a combinatorial embedding, it does not need to be planar.

In[1222]:=

```
emb = <|1 → {2, 4, 3}, 2 → {4, 3, 1}, 3 → {1, 2, 4}, 4 → {3, 1, 2}|>;
IGPlanarQ[emb]
```

Out[1223]=

```
False
```

In[1224]:=

```
IGDualGraph[emb]
```

Out[1224]=

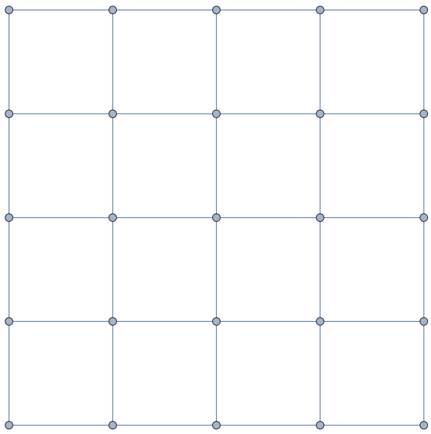


Find the dual of a square lattice graph. The dual graph also includes the outer face as a vertex.

In[1225]:=

```
IGSquareLattice[{5, 5}]
```

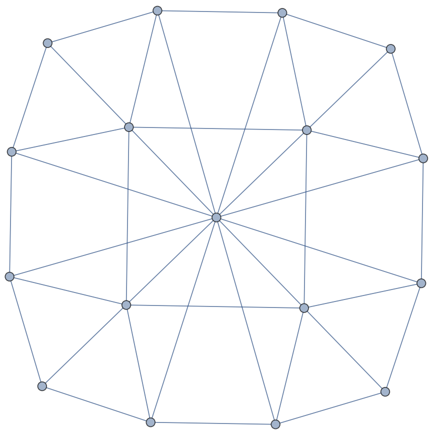
Out[1225]=



In[1226]:=

```
IGDualGraph[%]
```

Out[1226]=

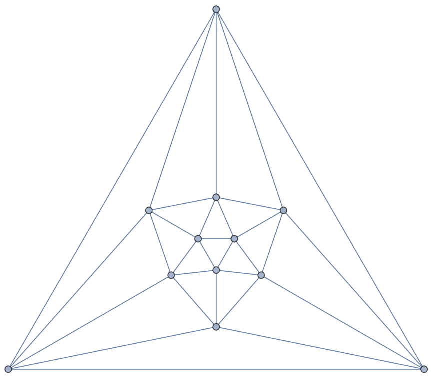


The dual is unique if the graph is 3-vertex-connected. This can be verified using `KVertexConnectedGraphQ`. In this case, `IGDualGraph@IGDualGraph[g]` is isomorphic to g.

In[1227]:=

```
g = GraphData["IcosahedralGraph"]
```

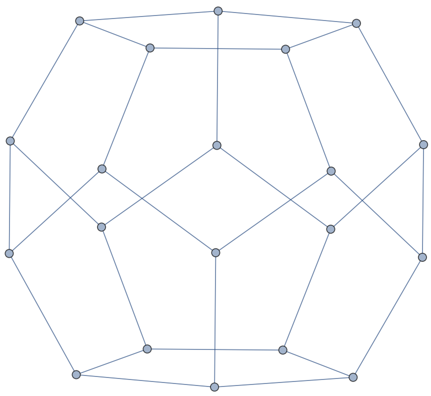
Out[1227]=



In[1228]:=

```
dg = IGDualGraph[g]
```

Out[1228]=



In[1229]:=

```
IGIsomorphicQ[dg, GraphData["DodecahedralGraph"]]
```

Out[1229]=

```
True
```

In[1230]:=

```
IGIsomorphicQ[IGDualGraph[dg], g]
```

Out[1230]=

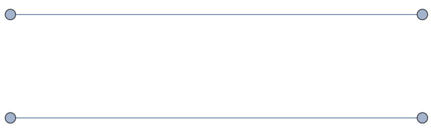
```
True
```

If the graph is not connected, the dual of each component is effectively computed separately.

In[1231]:=

```
IGDualGraph@IGDisjointUnion[{CycleGraph[3], CycleGraph[3]}]
```

Out[1231]=



## IGEmbeddingQ

In[1232]:=

```
? IGEmbeddingQ
```

IGEmbeddingQ[embedding] tests if embedding represents a combinatorial embedding of a simple graph.

`IGEmbeddingQ` checks if an embedding is valid, and whether it belongs to a graph without self-loops and multi-edges.

This is a valid combinatorial embedding of the graph 1 ⟷ 3 ⟷ 2.

In[1233]:=

```
IGEmbeddingQ[<|1 → {3}, 2 → {3}, 3 → {1, 2}|>]
```

Out[1233]=

```
True
```

The following embeddings do not belong to simple (i.e. loop free and multi-edge free) graphs:

In[1234]:=

```
IGEmbeddingQ[<|1 → {3}, 2 → {3, 3}, 3 → {1, 2, 2}|>]
```

Out[1234]=

```
False
```

In[1235]:=

```
IGEmbeddingQ[<|1 → {1, 2}, 2 → {1}|>]
```

Out[1235]=

```
False
```

The following embedding is not valid because it does not contain the arc 2 → 1 but it does contain 1 → 2.

In[1236]:=

```
IGEmbeddingQ[<|1 → {2}, 2 → {}|>]
```

Out[1236]=

```
False
```

## IGPlanarEmbedding

In[1237]:=

```
? IGPlanarEmbedding
```

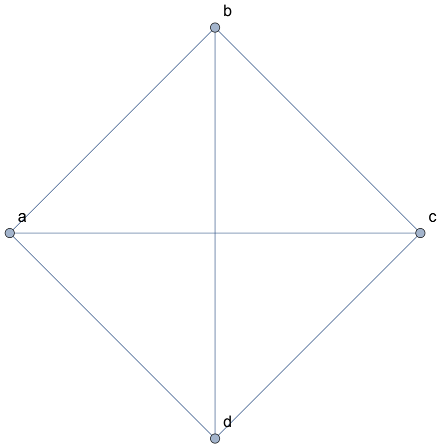
IGPlanarEmbedding[graph] gives a planar combinatorial embedding of a graph.

`IGPlanarEmbedding` computes a combinatorial embedding of a planar graph. The current implementation ignores self-loops and multi-edges.

In[1238]:=

```
g = IGShorthand["a:b:c:d -- a:b:c:d"]
```

Out[1238]=




In[1239]:=

```
emb = IGPlanarEmbedding[g]
```

Out[1239]=

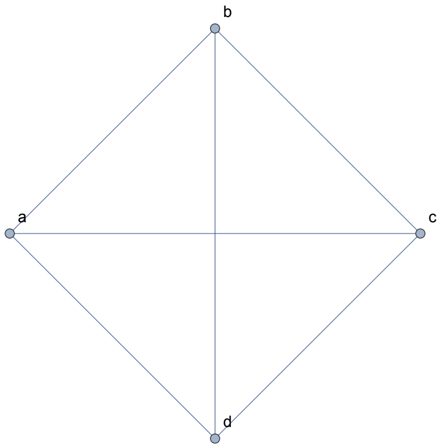
```
<|a → {b, c, d}, b → {a, d, c}, c → {b, d, a}, d → {c, b, a}|>
```

The representation of a combinatorial embedding is also a valid adjacency list, thus it can be easily converted back to an undirected graph using `IGAdjacencyGraph`.

In[1240]:=

```
IGAdjacencyGraph[emb, VertexLabels → Automatic]
```

Out[1240]=




## IGOuterplanarEmbedding

In[1241]:=

```
? IGOuterplanarEmbedding
```

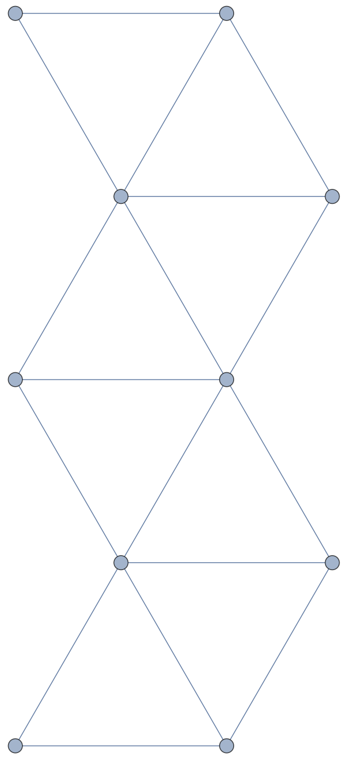
IGOuterplanarEmbedding[graph] gives an outerplanar combinatorial embedding of a graph.

`IGOuterplanarEmbedding` returns an outerplanar combinatorial embedding of a graph, if it exists. If the corresponding graph is connected, then one face of such an embedding contains all vertices of the graph.

In[1242]:=

```
g = IGTriangularLattice[{5, 2}]
```

Out[1242]=



In[1243]:=

```
emb = IGOuterplanarEmbedding[g]
```

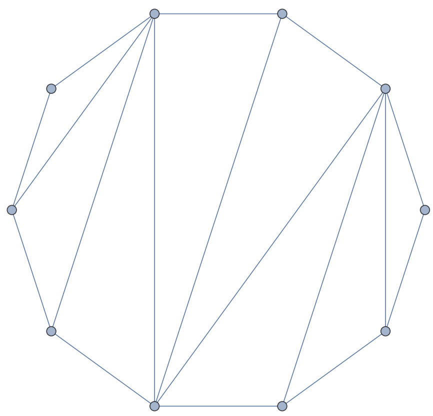
Out[1243]=

```
<|1 → {6, 2}, 2 → {1, 6, 7, 8, 3}, 3 → {2, 8, 4}, 4 → {3, 8, 9, 10, 5}, 5 → {4, 10},
 6 → {2, 1, 7}, 7 → {6, 8, 2}, 8 → {7, 9, 4, 3, 2}, 9 → {8, 10, 4}, 10 → {9, 5, 4}|>
```

In[1244]:=

```
IGLayoutCircle@IGReorderVertices[First@MaximalBy[Length]@IGFaces[emb], g]
```

Out[1244]=



## IGCoordinatesToEmbedding

In[1245]:=

```
? IGCoordinatesToEmbedding
```

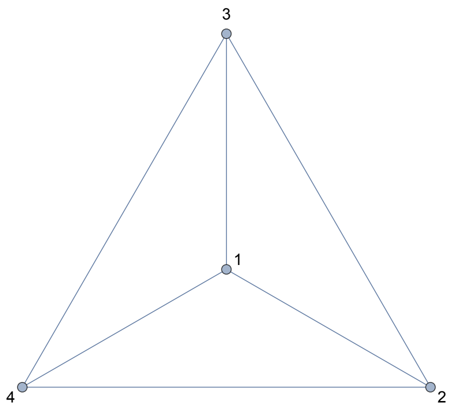
IGCoordinatesToEmbedding[graph] gives a combinatorial embedding based on the vertex coordinates of graph.
IGCoordinatesToEmbedding[graph, coordinates] uses the given coordinates instead of the VertexCoordinates property.

`IGCoordinatesToEmbedding` computes a combinatorial embedding, i.e. a cyclic ordering of neighbours around each vertex, based on the given vertex coordinates. By default, the coordinates are taken from the `VertexCoordinates` property.

In[1246]:=

```
g = CompleteGraph[4, VertexLabels → "Name"]
```

Out[1246]=




In[1247]:=

```
emb = IGCoordinatesToEmbedding[g]
```

Out[1247]=

```
<|1 → {4, 2, 3}, 2 → {3, 1, 4}, 3 → {4, 1, 2}, 4 → {2, 1, 3}|>
```

The embedding can then be used to compute the faces of the graph …

In[1248]:=

```
IGFaces[emb]
```

Out[1248]=

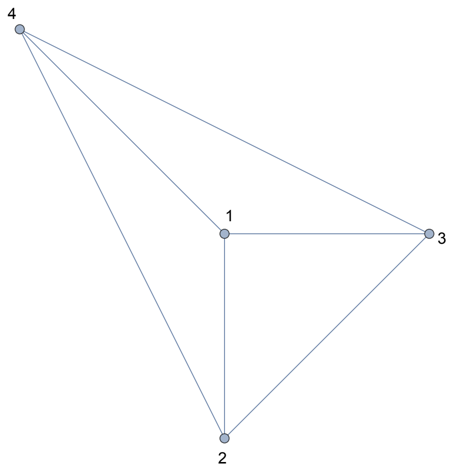
```
{{1, 4, 2}, {1, 2, 3}, {1, 3, 4}, {2, 4, 3}}
```

… or can be converted back to coordinates.

In[1249]:=

```
Graph[g, VertexCoordinates → IGEmbeddingToCoordinates[emb]]
```

Out[1249]=

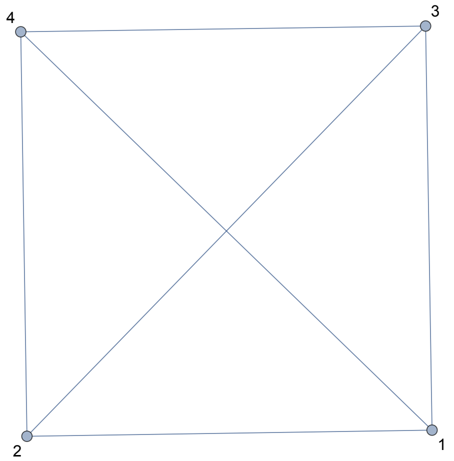



If we start with a non-planar graph layout, the embedding will not be planar either.

In[1250]:=

```
g = CompleteGraph[4, GraphLayout → "SpringElectricalEmbedding", VertexLabels → "Name"]
```

Out[1250]=




In[1251]:=

```
emb = IGCoordinatesToEmbedding[g]
```

Out[1251]=

```
<|1 → {2, 3, 4}, 2 → {1, 3, 4}, 3 → {4, 2, 1}, 4 → {2, 1, 3}|>
```

In[1252]:=

```
IGFaces[emb]
```

Out[1252]=

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

In[1253]:=

```
IGPlanarQ[emb]
```

Out[1253]=

```
False
```

## IGEmbeddingToCoordinates

In[1254]:=

```
? IGEmbeddingToCoordinates
```

IGEmbeddingToCoordinates[embedding] gives the
coordinates of a planar drawing based on the given combinatorial embedding.

`IGEmbeddingToCoordinates` computes the coordinates of a straight-line planar drawing based on the given combinatorial embedding, using Schnyder's algorithm.

In[1255]:=

```
emb1 = <|1 → {2, 3, 4}, 2 → {1, 4, 3}, 3 → {2, 4, 1}, 4 → {3, 2, 1}|>;
```

In[1256]:=

```
IGEmbeddingToCoordinates[emb1]
```

Out[1256]=

```
{{1, 1}, {1, 0}, {2, 1}, {0, 2}}
```

The embedding must be planar.

In[1257]:=

```
emb2 = <|1 → {2, 4, 3}, 2 → {4, 3, 1}, 3 → {1, 2, 4}, 4 → {3, 1, 2}|>;
```

In[1258]:=

```
IGPlanarQ[emb2]
```

Out[1258]=

```
False
```

In[1259]:=

```
IGEmbeddingToCoordinates[emb2]
```

IGraphM: embeddingToCoordinates: The embedding is not planar.

Out[1259]=

```
$Failed
```

## IGLayoutPlanar

In[1260]:=

```
? IGLayoutPlanar
```

IGLayoutPlanar[graph, options] lays out a planar graph using Schnyder's algorithm.

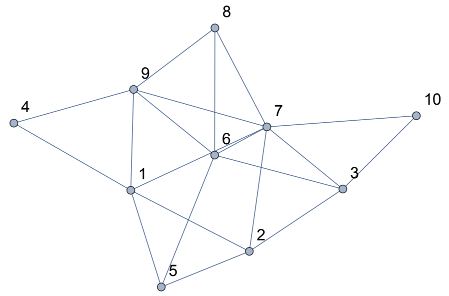
`IGLayoutPlanar` computes a layout of a planar graph without edge crossings using Schnyder's algorithm. The vertex coordinates will lie on an $(n-2)\times(n-2)$ integer grid, where $n$ is the number of vertices.

Create a random planar graph and lay it out without edge crossings.

In[1261]:=

```
g = IGTryUntil[IGPlanarQ]@RandomGraph[{10, 20}, VertexLabels → "Name"]
```

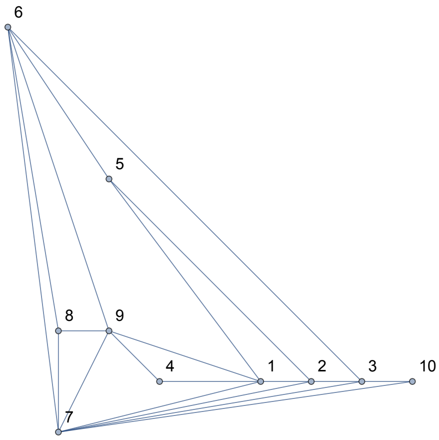
Out[1261]=



In[1262]:=

```
IGLayoutPlanar[g]
```

Out[1262]=




`IGLayoutPlanar` produces a drawing based on the combinatorial embedding returned `IGPlanarEmbedding`. A combinatorial embedding is a counter-clockwise ordering of the incident edges around each vertex.

In[1263]:=

```
emb = IGPlanarEmbedding[g]
```

Out[1263]=

```
<|1 → {2, 5, 9, 4, 7}, 2 → {1, 7, 3, 5}, 3 → {2, 7, 10, 6}, 4 → {1, 9}, 5 → {2, 6, 1},
 6 → {5, 3, 7, 8, 9}, 7 → {6, 10, 3, 2, 1, 9, 8}, 8 → {7, 9, 6}, 9 → {8, 7, 4, 1, 6}, 10 → {7, 3}|>
```

The embedding can also be used to directly compute coordinates for a drawing.

In[1264]:=

```
IGEmbeddingToCoordinates[emb]
```

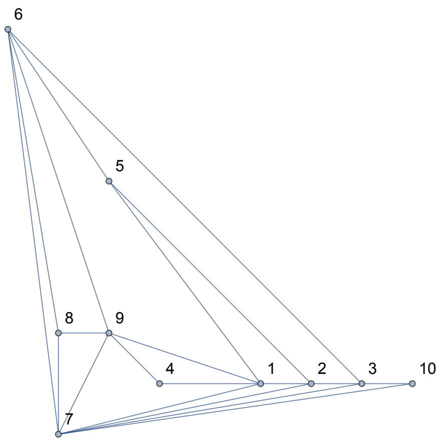
Out[1264]=

```
{{5, 1}, {6, 1}, {7, 1}, {3, 1}, {2, 5}, {0, 8}, {1, 0}, {1, 2}, {2, 2}, {8, 1}}
```

In[1265]:=

```
Graph[g, VertexCoordinates → %]
```

Out[1265]=




## IGLayoutTutte

In[1266]:=

```
? IGLayoutTutte
```

IGLayoutTutte[graph, options] lays out a 3–vertex–connected planar graph using the Tutte embedding.

The Tutte embedding can be computed for a 3-vertex-connected planar graph. The faces of such a graph are uniquely defined. This embedding ensures that the coordinates of any vertex not on the outer face are the average of its neighbour's coordinates, thus it is also called barycentric embedding.

IGLayoutTutte supports weighted graphs, and uses the weights for computing barycentres.

The available options are:

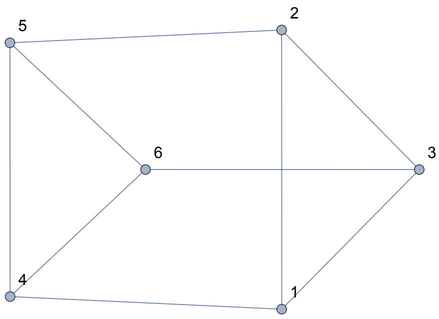
- "OuterFace" sets the planar graph face to use as the outer face for the layout. The vertices of the face can be given in any order. Use IGFaces to obtain a list of faces.

By default, a largest face is chosen to be the outer one.

In[1267]:=

```
g = IGShorthand["1-2-3-1,4-5-6-4,1-4,2-5,3-6"]
```

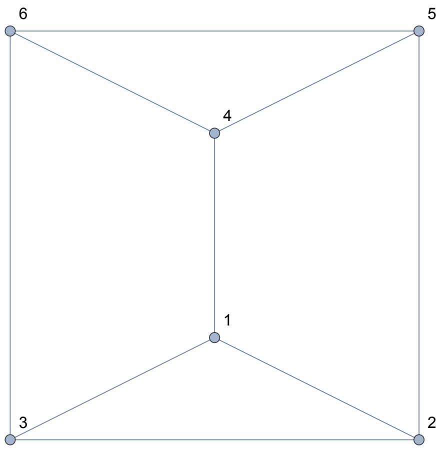
Out[1267]=




In[1268]:=

```
IGLayoutTutte[g]
```

Out[1268]=

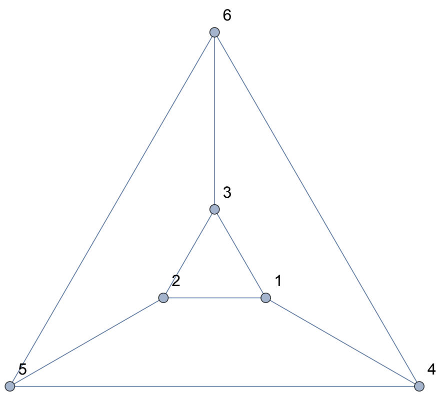



We can specify a different outer face manually.

In[1269]:=

```
IGLayoutTutte[g, "OuterFace" → {5, 4, 6}]
```

Out[1269]=

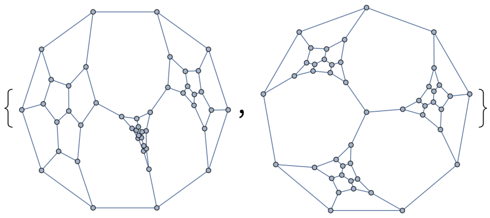


For some graphs, the best result is achieved when the outer face is not chosen to be a largest one.

In[1270]:=
```
g = GraphData["TutteGraph"];
```

In[1271]:=
```
{IGLayoutTutte[g], IGLayoutTutte[g, "OuterFace" → {2, 8, 9, 10, 7, 6, 5, 4, 3}]}
```

Out[1271]=



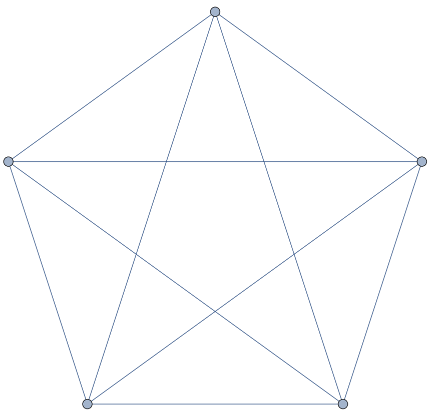
`IGLayoutTutte` requires a 3-vertex-connected planar input.

In[1272]:=
```
IGLayoutTutte[CompleteGraph[5]]
```

IGraphM: planarEmbedding: The graph is not planar.

IGLayoutTutte : The graph is not planar and a Tutte embedding cannot be computed. Vertex coordinates will not be set.

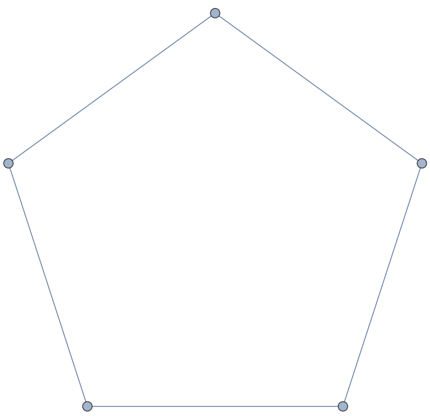
Out[1272]=



In[1273]:=
```
IGLayoutTutte[CycleGraph[5]]
```

IGLayoutTutte : The graph is not 3–vertex–connected and a Tutte embedding cannot be computed. Vertex coordinates will not be set.

Out[1273]=



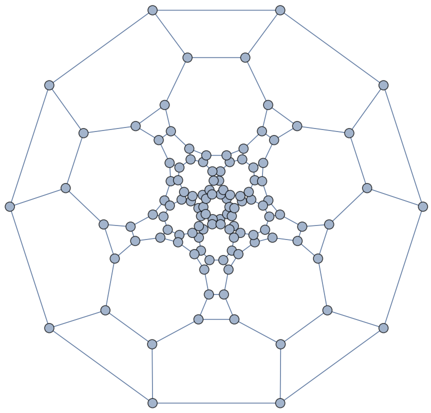
`IGLayoutTutte` will take into account edge weights. For a weighted graph, the barycenter of neighbours is computed with a weighting corresponding to the edge weights.

A disadvantage of the Tutte embedding is that the ratio of the shortest and longest edge it creates is often very large. This can be partially remedied by first computing an unweighted Tutte embedding, then setting edge weights based on the obtained edge lengths.

In[1274]:=

```
pg = IGLayoutTutte@GraphData["GreatRhombicosidodecahedralGraph"]
```

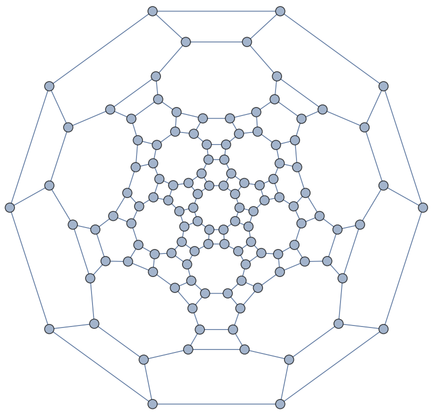
Out[1274]=



In[1275]:=

```
IGLayoutTutte@
 IGEdgeMap[Apply[EuclideanDistance], EdgeWeight → IGEdgeVertexProp[VertexCoordinates], pg]
```

Out[1275]=

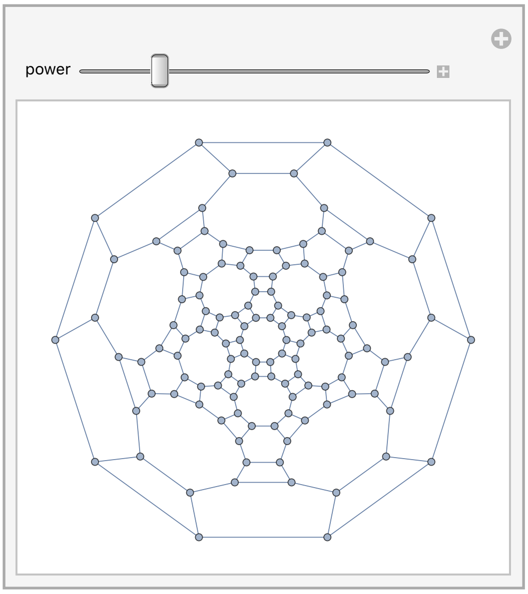


By applying a further power-transformation of the weight, we can fine-tune the layout.

In[1276]:=

```
Manipulate[
 IGLayoutTutte[
  IGEdgeMap[(EuclideanDistance @@ #) ^ power &,
   EdgeWeight → IGEdgeVertexProp[VertexCoordinates], pg],
  VertexSize → 1 / 2
 ],
 {{power, 1}, 0.5, 3},
 Initialization :→ Needs["IGraphM`"]
]
```

Out[1276]=




# Geometrical computation and meshes

## Geometrical meshes

### IGMeshGraph

In[1277]:=

```
? IGMeshGraph
```

IGMeshGraph[mesh] converts the edges and vertices of a geometrical mesh to a weighted graph.

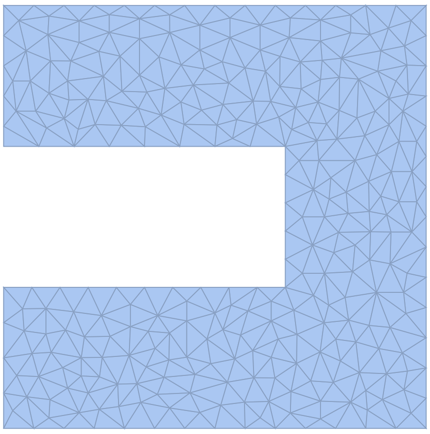
The available options are:

- `EdgeWeight` sets either the explicit edge weights, or the mesh property to be used as edge weights. The default value is `MeshCellMeasure`. Use None to obtain an unweighted graph.

The following example demonstrates finding a shortest path on a geometric mesh.

In[1278]:=
```
mesh = DiscretizeRegion[
  RegionDifference[Rectangle[{0, 0}, {3, 3}], Rectangle[{0, 1}, {2, 2}]], MaxCellMeasure → 0.02]
```

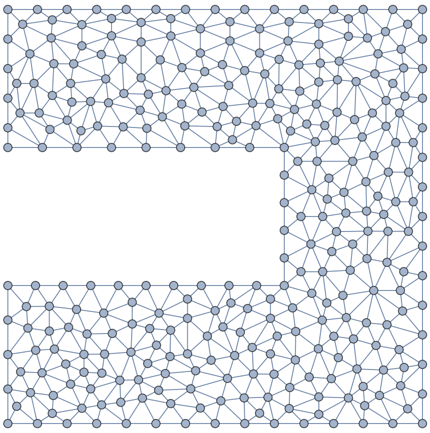
Out[1278]=



`IGMeshGraph` preserves the vertex coordinates, and uses edge lengths as edge weights by default.

In[1279]:=
```
g = IGMeshGraph[mesh]
```

Out[1279]=



Find the corners.

In[1280]:=
```
st = First /@ Through[
   {MinimalBy, MaximalBy}[VertexList[g], Norm@PropertyValue[{g, #}, VertexCoordinates] &]
  ]
```

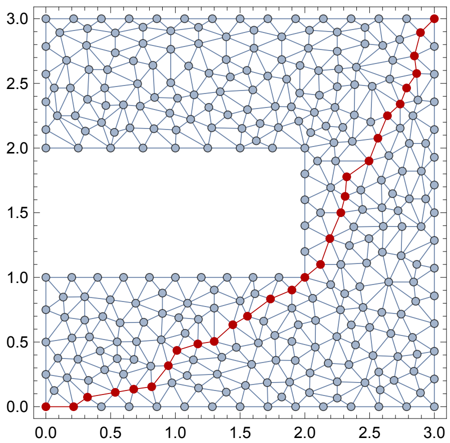
Out[1280]=
```
{48, 20}
```

Highlight the shortest path.

In[1281]:=

```
HighlightGraph[g,
 PathGraph@FindShortestPath[g, First[st], Last[st]],
 Frame → True, FrameTicks → True
]
```

Out[1281]=

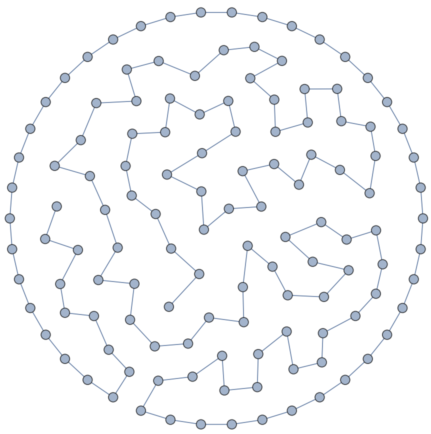



Find a Hamiltonian path on a mesh.

In[1282]:=

```
g = IGMeshGraph@DiscretizeRegion[Disk[], MaxCellMeasure → 1 / 40];
HighlightGraph[g, PathGraph@FindHamiltonianPath[g], GraphHighlightStyle → "DehighlightHide"]
```

Out[1283]=

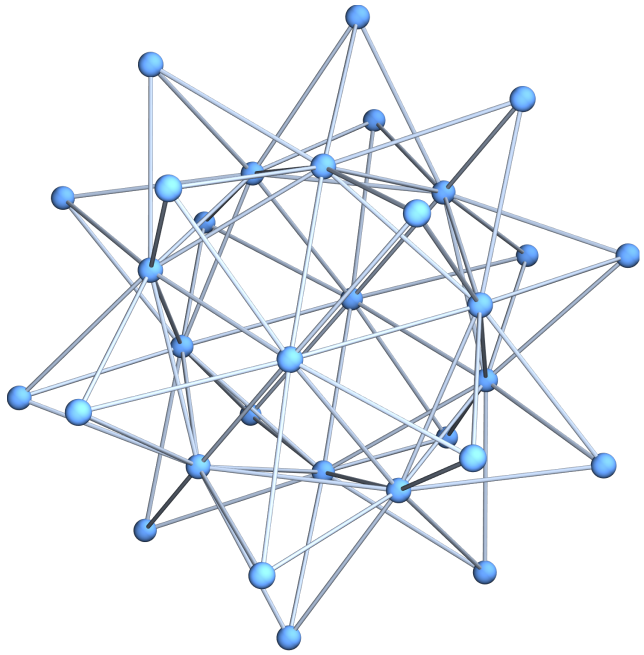


Get a spikey as a graph.

In[1284]:=

```
IGMeshGraph@PolyhedronData["Spikey", "BoundaryMeshRegion"]
```

Out[1284]=



## IGMeshCellAdjacencyGraph and IGMeshCellAdjacencMatrix

In[1285]:=

```
? IGMeshCellAdjacencyGraph
```

IGMeshCellAdjacencyGraph[mesh, d] gives the connectivity structure of d–dimensional cells in mesh as a graph.
IGMeshCellAdjacencyGraph[mesh, d1, d2] gives the
connectivity structure of d1 and d2 dimensional cells in mesh as a bipartite graph.

In[1286]:=

```
? IGMeshCellAdjacencyMatrix
```

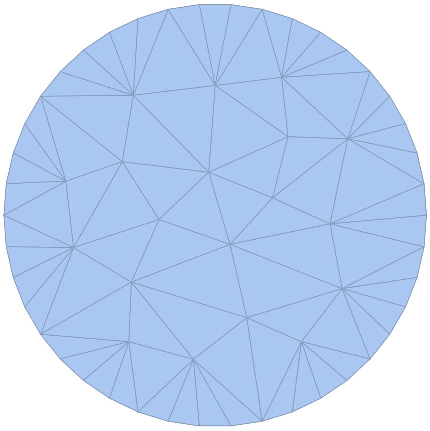
IGMeshCellAdjacencyMatrix[mesh, d] gives the adjacency matrix of d–dimensional cells in mesh.
IGMeshCellAdjacencyMatrix[mesh, d1, d2] gives the incidence matrix of d1– and d2–dimensional cells in mesh.

The available options for `IGMeshCellAdjacencyGraph` are:

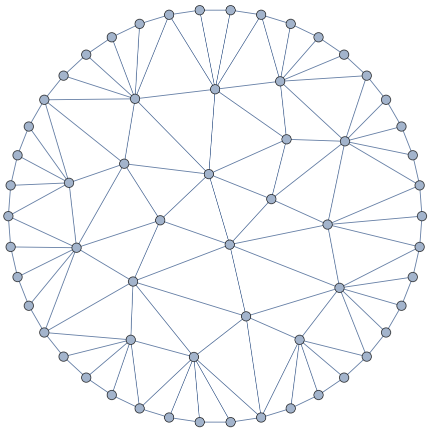
- `VertexCoordinates → Automatic` will use the mesh cell centroids as vertex coordinates if the mesh is 2 or 3-dimensional. The default is `VertexCoordinates → None`, which does not compute any coordinates.

Compute the connectivity of mesh vertices (zero-dimensional cells).

In[1287]:=
```
mesh = DiscretizeRegion[Disk[], MaxCellMeasure → 0.1]
```
Out[1287]=

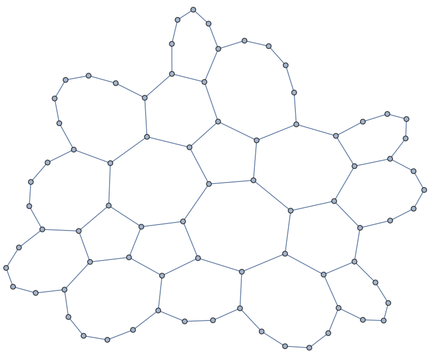


In[1288]:=
```
IGMeshCellAdjacencyGraph[mesh, 0, VertexCoordinates → Automatic]
```
Out[1288]=



Compute the connectivity of faces (two-dimensional cells).

In[1289]:=
```
IGMeshCellAdjacencyGraph[mesh, 2]
```
Out[1289]=

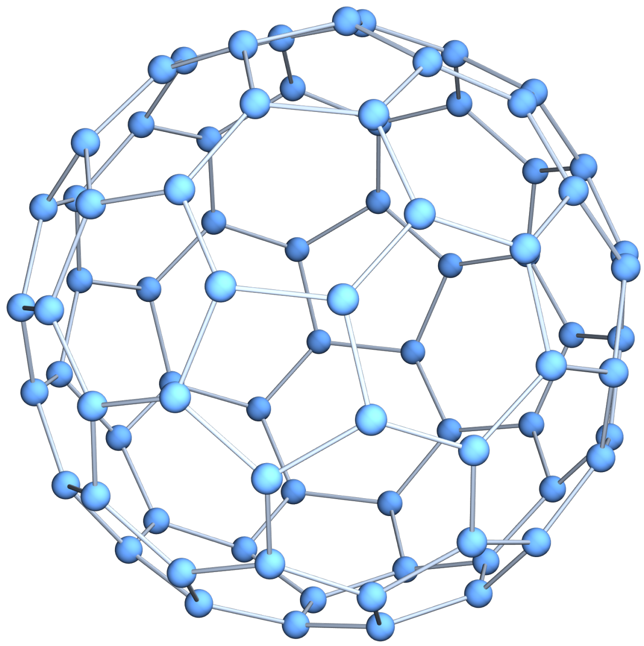


Create the graph of a Goldberg polyhedron.

In[1290]:=

```
IGMeshCellAdjacencyGraph[
 BoundaryDiscretizeRegion[Ball[], PrecisionGoal → 1, MaxCellMeasure → 0.5], 2,
 VertexCoordinates → Automatic]
```

Out[1290]=

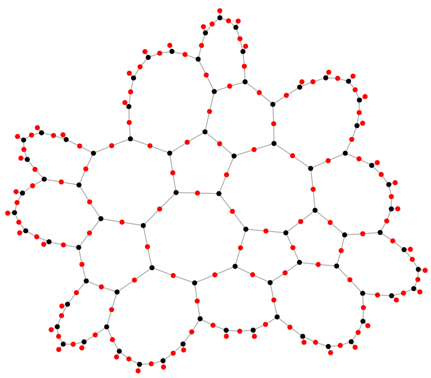


Compute the connectivity of faces and edges, and colour nodes based on whether they represent a face or an edge.

In[1291]:=

```
g = IGMeshCellAdjacencyGraph[
  mesh, 2, 1,
  VertexSize → 0.9, VertexStyle → {EdgeForm[], {1, _} → Red, {2, _} → Black}, EdgeStyle → Gray]
```

Out[1291]=



This is a bipartite graph.

In[1292]:=

```
IGBipartiteQ[g]
```

Out[1292]=

```
True
```

The vertex names are the same as the mesh cell indices (see `MeshCellIndex`).

In[1293]:=

```
VertexList[g] // Short
```

Out[1293]//Short=

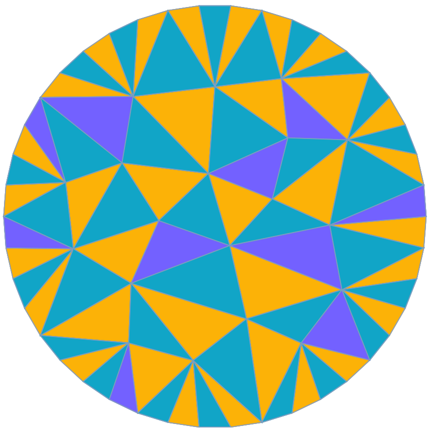
```
{{2, 1}, {2, 2}, {2, 3}, {2, 4}, {2, 5}, {2, 6},
 <<204>>, {1, 133}, {1, 134}, {1, 135}, {1, 136}, {1, 137}, {1, 138}}
```

Colour the faces of the mesh.

In[1294]:=

```
SetProperty[{mesh, {2, All}},
 MeshCellStyle → ColorData[100] /@ IGVertexColoring@IGMeshCellAdjacencyGraph[mesh, 2]
]
```

Out[1294]=



The edge-edge connectivity is identical to the line graph of the vertex-vertex connectivity.

In[1295]:=

```
IGIsomorphicQ[
 LineGraph@IGMeshCellAdjacencyGraph[mesh, 0], IGMeshCellAdjacencyGraph[mesh, 1]
]
```

Out[1295]=

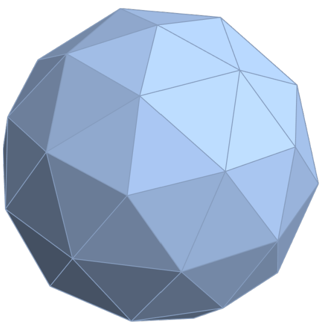
```
True
```

Compute the adjacency matrix of the vertex-vertex connectivity.

In[1296]:=

```
mesh = DiscretizeRegion[Sphere[],
   PrecisionGoal → 1.5, MaxCellMeasure → 1, ImageSize → Small]
```

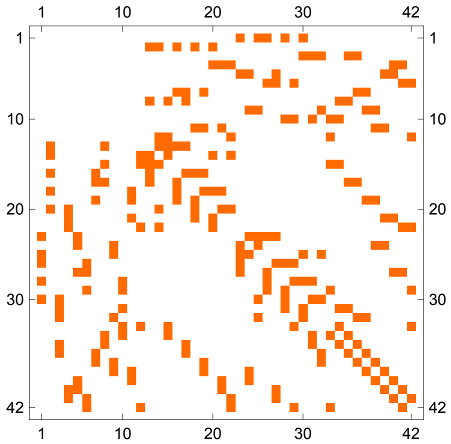
Out[1296]=



In[1297]:=

```
MatrixPlot@IGMeshCellAdjacencyMatrix[mesh, 0]
```

Out[1297]=




Compute the adjacency matrix of the edge-face connectivity.

In[1298]:=

```
bm = IGMeshCellAdjacencyMatrix[mesh, 1, 2]
```

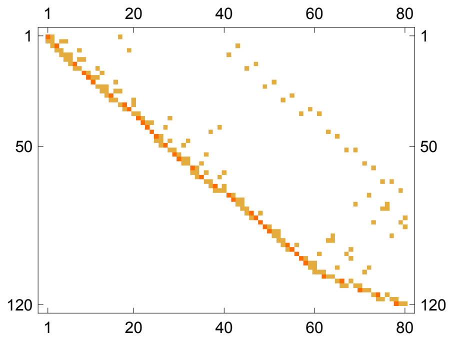
Out[1298]=

```
SparseArray[ Specified elements: 240
             Dimensions: {120, 80} ]
```

In[1299]:=

```
MatrixPlot[bm]
```

Out[1299]=

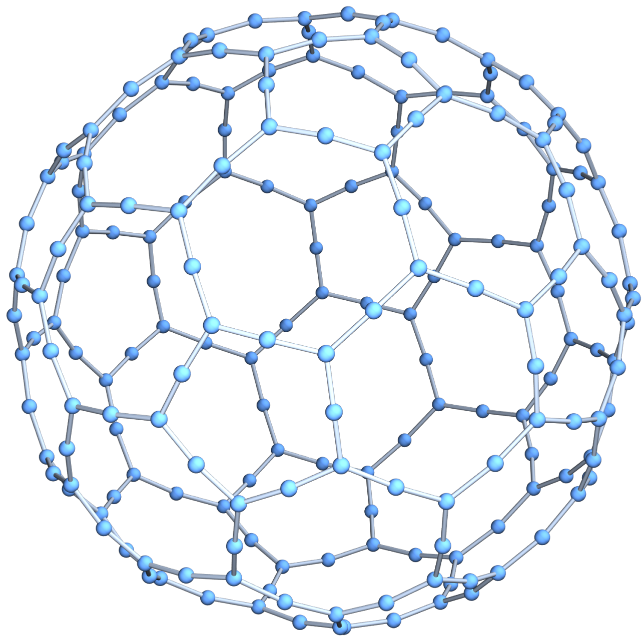


This is the (non-square) incidence matrix of a bipartite graph. The graph can be reconstructed using `IGBipartiteIncidenceGraph`.

In[1300]:=

```
Graph3D@IGBipartiteIncidenceGraph[bm]
```

Out[1300]=



Paint a Hamiltonian path on triangulation using a gradient of colours.

In[1301]:=

```
mesh = DiscretizeRegion[Disk[], MaxCellMeasure → 1 / 50, MeshCellStyle → {1 → None}];
path = FindHamiltonianPath@IGMeshCellAdjacencyGraph[mesh, 2];
```

In[1303]:=

```
MeshRegion[
 mesh,
 MeshCellStyle → MapIndexed[#1 → ColorData["Pastel"][First[#2] / Length[path]] &, path]
]
```

Out[1303]=

## IGLatticeMesh

In[1304]:=

```
? IGLatticeMesh
```

IGLatticeMesh[type] creates a mesh of the lattice of the specified type.
IGLatticeMesh[type, {m, n}] creates a lattice of n by m unit cells.
IGLatticeMesh[type, region] creates a lattice from the points that fall within region.
IGLatticeMesh[] gives a list of available lattice types.

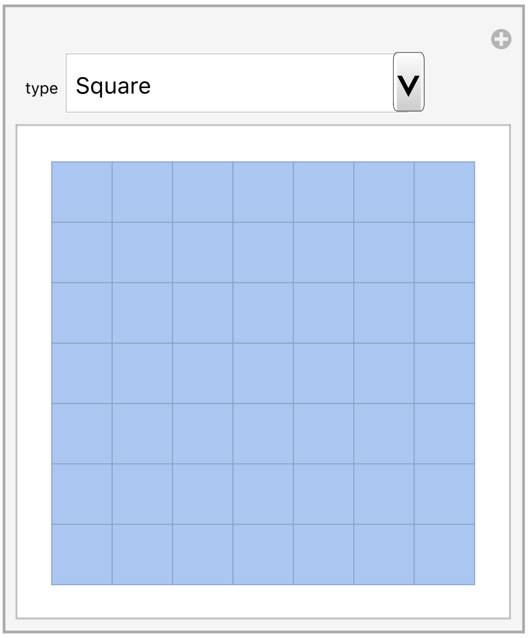
`IGLatticeMesh` can generate meshes of various periodic tilings. `IGMeshGraph` and `IGMeshCellAdjacencyGraph` can be used to convert these to graphs. The primary use case is the easy generation of various lattice graphs.

`IGLatticeMesh[]` returns the list of available lattices. Let us explore them using a graphical interface.

In[1305]:=

```
Manipulate[IGLatticeMesh[type], {type, IGLatticeMesh[]}, Initialization :> Needs["IGraphM`"]]
```

Out[1305]=



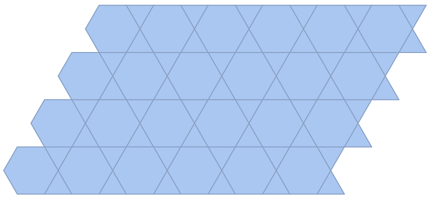

IGraph/M knows about a subset of the tilings available in `EntityClass["PeriodicTiling", All]`. Use these entities to obtain additional geometric information about the tilings.

Generate a *kagome lattice* consisting of 6 by 4 unit cells.

In[1306]:=

```
IGLatticeMesh["Trihexagonal", {6, 4}]
```

Out[1306]=

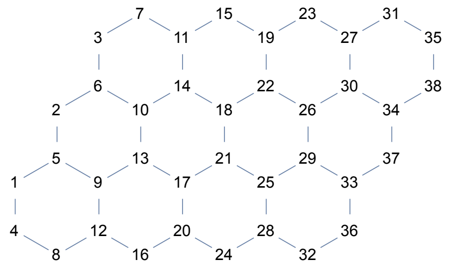


Create a hexagonal graph of 4 by 3 cells. Notice that the nodes are labelled with consecutive integers along the translation vectors of the lattice.

In[1307]:=

```
IGMeshGraph[
 IGLatticeMesh["Hexagonal", {4, 3}],
 VertexShapeFunction → "Name", PerformanceGoal → "Quality"]
```

Out[1307]=

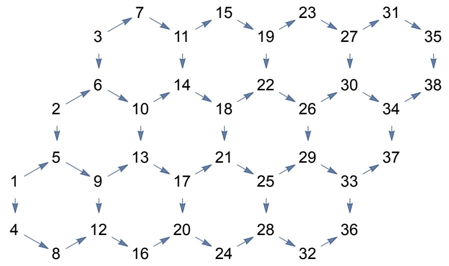



This specific node labelling allows for the creation of convenient directed lattices.

In[1308]:=

```
DirectedGraph[%, "Acyclic"]
```

Out[1308]=

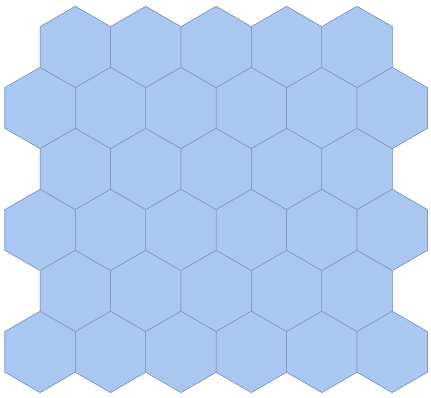



Create a hexagonal mesh from points that fall within a rectangular region.

In[1309]:=

```
IGLatticeMesh["Hexagonal", Rectangle[{0, 0}, {5, 5}]]
```

Out[1309]=

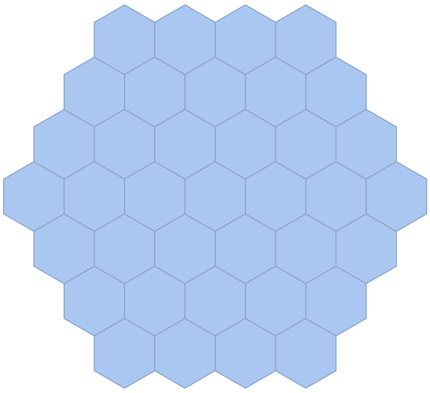


Create a hexagonal mesh from points that fall within a hexagonal region.

In[1310]:=

```
IGLatticeMesh["Hexagonal", Polygon@CirclePoints[3, 6]]
```

Out[1310]=

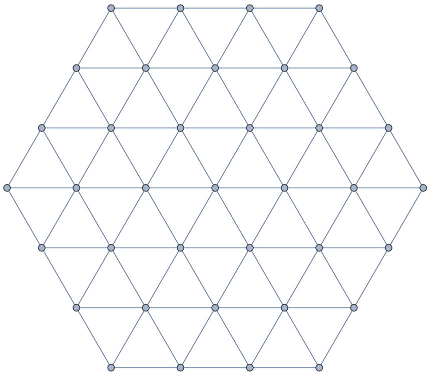


Create a triangular grid graph in the shape of a hexagon, as the face-adjacency graph of the above mesh.

In[1311]:=

```
IGMeshCellAdjacencyGraph[%, 2, VertexCoordinates → Automatic]
```

Out[1311]=



Create a face adjacency graph of the Cairo pentagonal tiling, and display it along with its mesh.

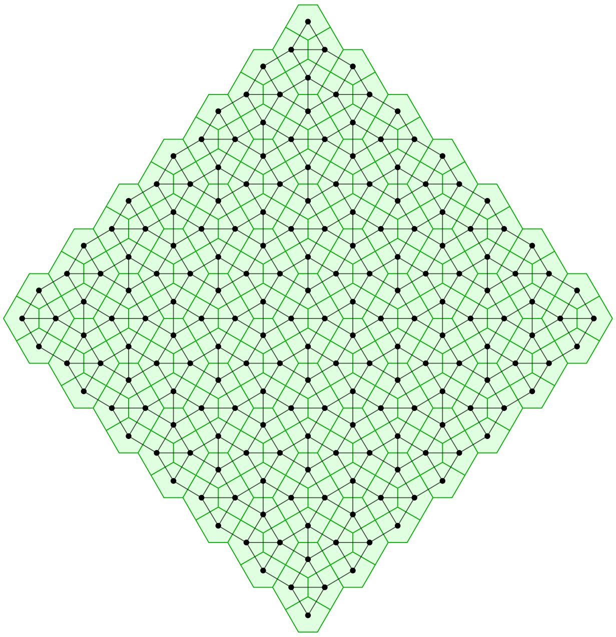
In[1312]:=

```
mesh = IGLatticeMesh["CairoPentagonal", MeshCellStyle → {1 → Darker@Green, 2 → LightGreen}];
```

In[1313]:=

```
Show[
 mesh,
 IGMeshCellAdjacencyGraph[mesh, 2,
  VertexCoordinates → Automatic, GraphStyle → "BasicBlack"],
 ImageSize → Medium
]
```

Out[1313]=



Compute a colouring of a periodic tiling so that neighbouring cells have different colours.

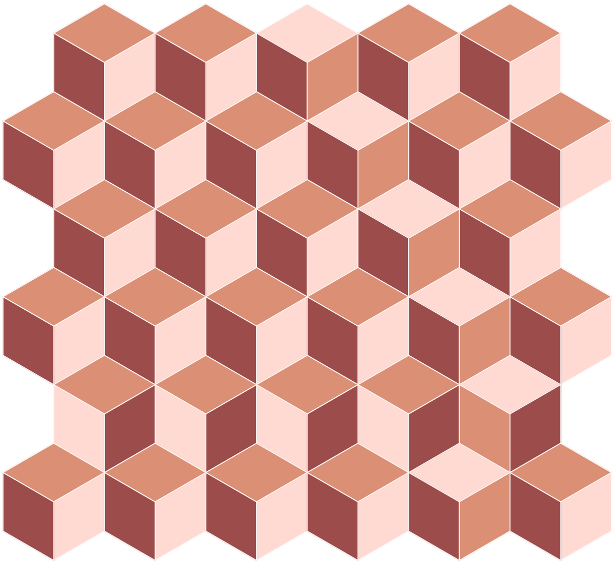
In[1314]:=

```
colorMesh[mesh_] :=
 SetProperty[{MeshRegion[mesh, MeshCellStyle → {1 → White}], {2, All}},
  MeshCellStyle → ColorData[8] /@ IGMinimumVertexColoring@IGMeshCellAdjacencyGraph[mesh, 2]
 ]
```

In[1315]:=

```
colorMesh@IGLatticeMesh["Rhombille", Rectangle[{0, 0}, {10, 10}], ImageSize → Medium]
```

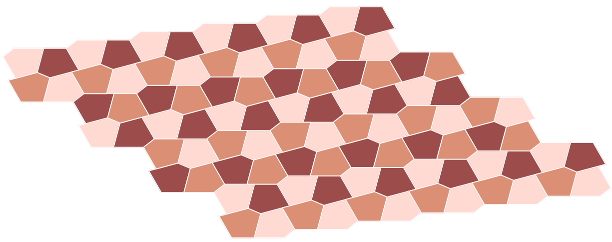
Out[1315]=



In[1316]:=

```
colorMesh@IGLatticeMesh["PentagonType2", {6, 4}, ImageSize → Medium]
```

Out[1316]=

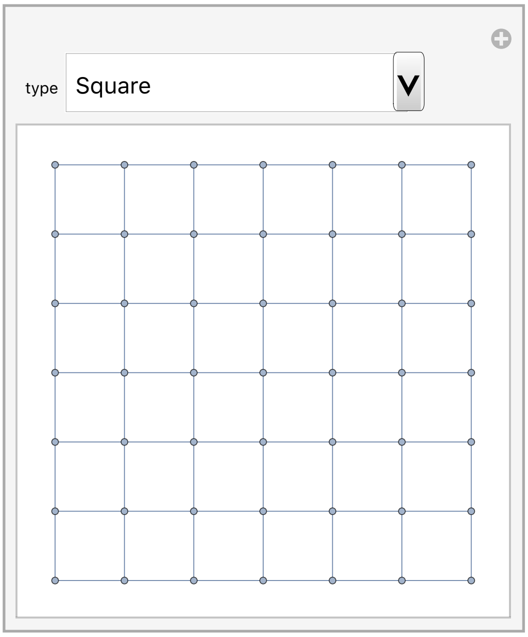


Explore the face-adjacency graphs of lattices. These correspond to the dual lattice.

In[1317]:=

```
Manipulate[
 IGMeshCellAdjacencyGraph[IGLatticeMesh[type], 2, VertexCoordinates → Automatic],
 {type, IGLatticeMesh[]}, Initialization :→ Needs["IGraphM`"]
]
```

Out[1317]=




Make a maze through the faces of a lattice. We start by finding a spanning tree of the face-edge incidence graph of the lattice.

In[1318]:=

```
mesh = IGLatticeMesh["Square", Disk[{0, 0}, 9.5], MeshCellStyle → {1 | 2 → GrayLevel[0.9]}];
t = IGRandomSpanningTree@IGMeshCellAdjacencyGraph[mesh, 2, 1];
```

The walls of the maze will be the leaves of this tree which are edges.

In[1320]:=

```
walls = Cases[Pick[VertexList[t], VertexDegree[t], 1], {1, _}];
```

We will remove two outer walls to serve as the

In[1321]:=

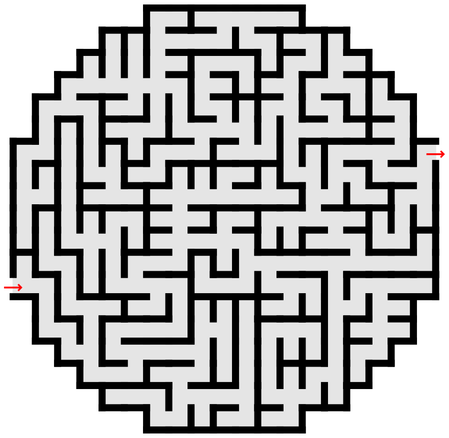
```
exits = SortBy[walls, PropertyValue[{mesh, #}, MeshCellCentroid].{1, 0.1} &][[{1, -1}]];
```

Draw the maze.

In[1322]:=

```
MeshRegion[mesh,
 MeshCellStyle → Thread[Complement[walls, exits] → Directive[AbsoluteThickness[4], Black]],
 Epilog → {Text["⟶", #] & /@ PropertyValue[{mesh, exits}, MeshCellCentroid]}
]
```

Out[1322]=

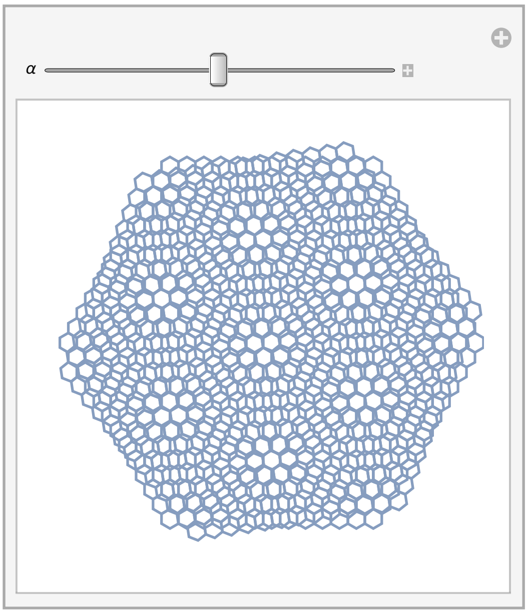


Create a Moiré pattern by superimposing two rotated hexagonal lattices.

In[1323]:=

```
m = IGLatticeMesh["Hexagonal", Polygon@CirclePoints[12., 6]];
Manipulate[
 Show@Table[
   MeshRegion[
    TransformedRegion[m, RotationTransform[angle]],
    MeshCellStyle → {2 → None, 1 → AbsoluteThickness[1.5]}, PlotRange → 13 {{-1, 1}, {-1, 1}}
   ],
   {angle, {0, α}}
  ],
 {{α, 0.15}, 0, 0.3},
 Initialization :→ Needs["IGraphM`"]
]
```

Out[1324]=




## Proximity graphs

Proximity graphs are connectivity structures of geometric points based on geometric criteria. IGraph/M implements several proximity graphs for points in two-dimensional Euclidean space.

### IGDelaunayGraph

In[1325]:=

```
? IGDelaunayGraph
```

IGDelaunayGraph[points] gives the Delaunay graph of the given points.

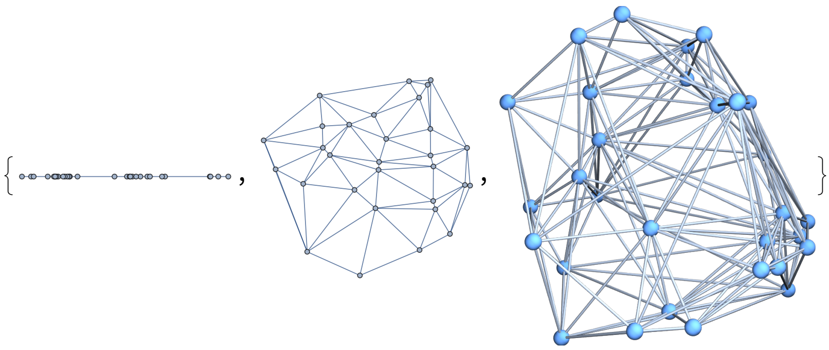
`IGDelaunayGraph[points]` creates computes the Delaunay graph of the given points in one, two or three dimensions. It is equivalent to `IGMeshGraph@DelaunayMesh[points]`, but it is faster and it supports collinear points in 2D and coplanar points in 3D.

`IGDelaunayGraph` works in 1D, 2D and 3D.

In[1326]:=

```
Table[
 IGDelaunayGraph@RandomPoint[Ball@ConstantArray[0, dim], 30],
 {dim, 1, 3}
]
```

Out[1326]=

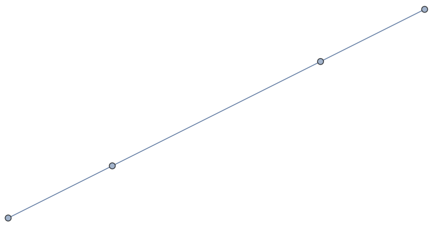


IGDelaunayGraph works with collinear points in 2D ...

In[1327]:=

```
IGDelaunayGraph[{{0, 0}, {2, 1}, {8, 4}, {6, 3}}]
```

Out[1327]=

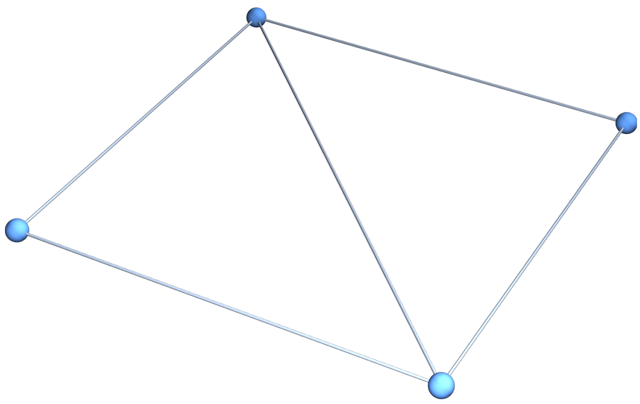


... or coplanar points in 3D.

In[1328]:=

```
IGDelaunayGraph[{{0, 0, 0}, {1, 0, 0}, {0, 1, 0}, {1, 1, 0}}]
```

Out[1328]=

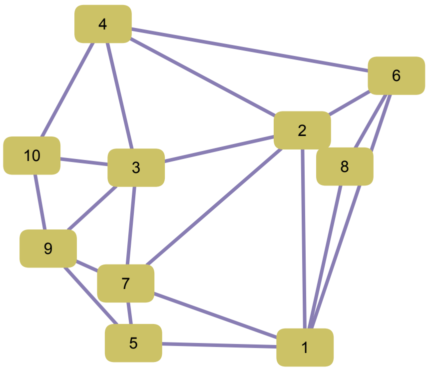


IGDelaunayGraph takes all the usual graph options.

In[1329]:=

```
IGDelaunayGraph[RandomReal[1, {10, 2}], GraphStyle → "DiagramGold"]
```

Out[1329]=

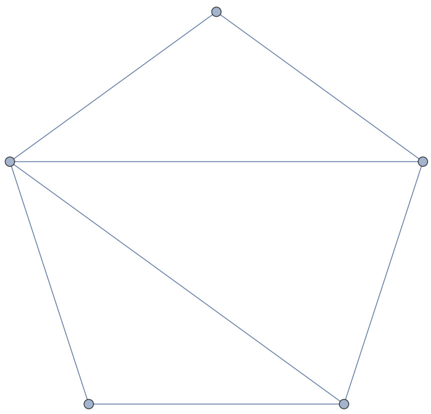



When there is more than one valid Delaunay triangulation, only one is returned.

In[1330]:=

```
IGDelaunayGraph@CirclePoints[5]
```

Out[1330]=



Find and plot an Euclidean minimum spanning tree of a set of points in three dimensions.

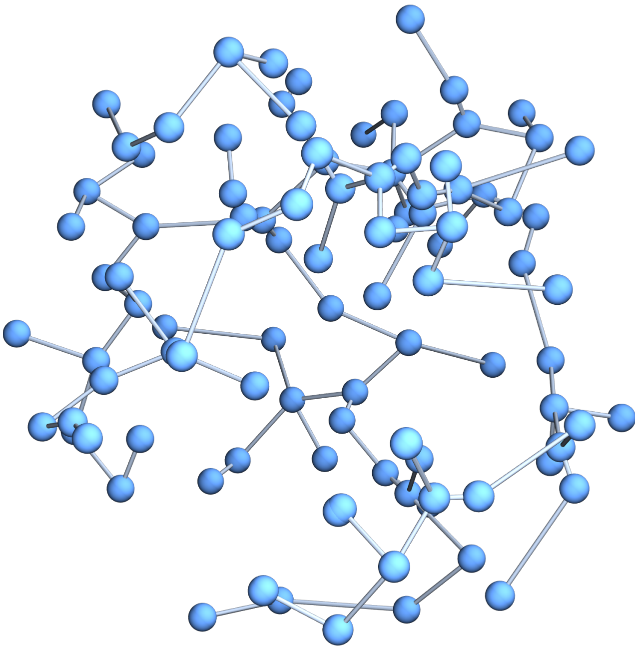
In[1331]:=

```
pts = RandomPoint[Ball[], 100];
```

In[1332]:=

```
dg = IGDelaunayGraph[pts];
IGTakeSubgraph[dg,
 IGSpanningTree@
  IGEdgeMap[Apply[EuclideanDistance], EdgeWeight → IGEdgeVertexProp[VertexCoordinates], dg]
]
```

Out[1333]=



## IGLuneBetaSkeleton

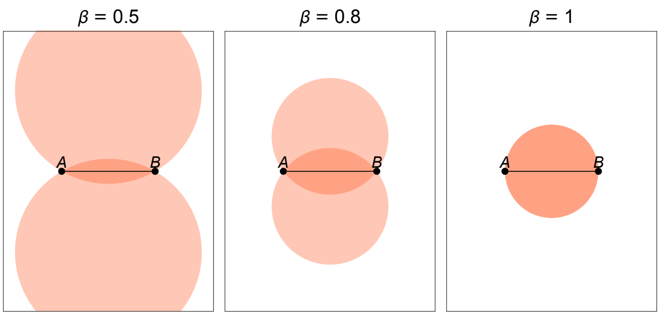
In[1334]:=

```
? IGLuneBetaSkeleton
```

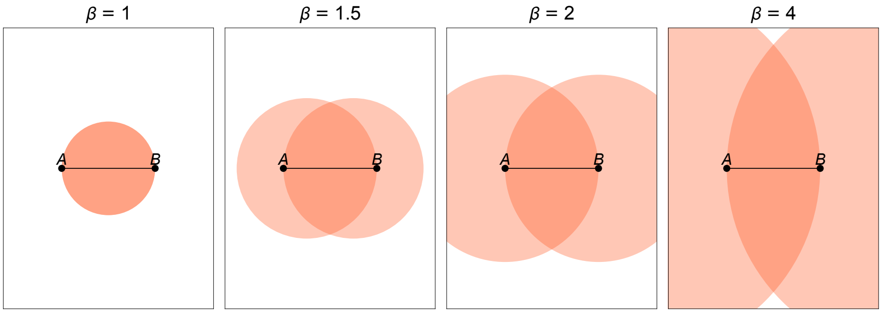
IGLuneBetaSkeleton[points, beta] gives the lune–based beta skeleton of the given points.

The lune-based $\beta$ skeleton connects two points $A$ and $B$ when the intersection of two disks (a lune) having $A$ and $B$ on its boundary contains no other points.

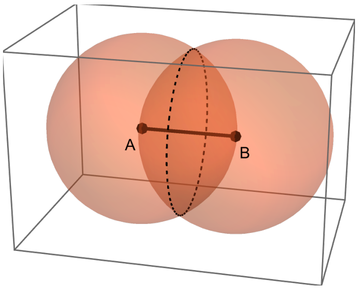
For $\beta \leq 1$, the lune is defined by disks of radius $AB/(2\beta)$.




For $\beta \geq 1$, the lune is defined by disks of radius $\beta AB/2$.




For $\beta \geq 1$, the definition generalizes to higher dimensions too. `IGLuneBetaSkeleton` supports 2D and 3D point sets.




For $\beta \geq 1$, the $\beta$ skeleton is a subgraph of the Delaunay graph, thus its edges do not cross. For $\beta < 2$, it contains the Euclidean minimum spanning tree, thus it is connected. For $\beta > 2$, it is typically disconnected. For $\beta = 2$, it may be disconnected in special degenerate cases when three neighbouring points form an equilateral triangle.

The implementation of $\beta$ skeleton computation is efficient only for $\beta \geq 1$.

$\beta$ skeletons can be used to reconstruct a shape from a set of points.

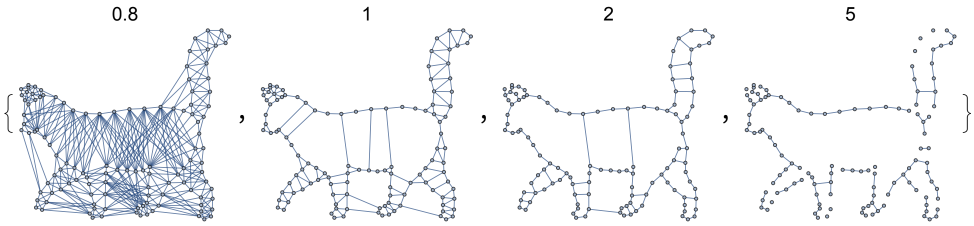
In[1335]:=

```
points = {{2.21, 3.83}, {2.5, 3.59}, {2.9, 3.49}, {3.33, 3.48}, {3.79, 3.55}, {4.27, 3.63},
   {4.74, 3.65}, {5.25, 3.7}, {5.66, 3.65}, {5.98, 3.58}, {6.25, 3.47}, {6.43, 3.23}, {6.47, 2.86},
   {6.35, 2.39}, {6.19, 2.02}, {5.98, 1.77}, {5.8, 1.57}, {5.51, 1.31}, {5.32, 0.94}, {5.19, 0.59},
   {5., 0.42}, {4.83, 0.31}, {4.62, 0.28}, {4.5, 0.37}, {4.54, 0.51}, {4.7, 0.58}, {4.82, 0.68},
   {4.87, 0.91}, {4.87, 1.17}, {4.9, 1.6}, {4.91, 1.81}, {4.7, 1.7}, {4.47, 1.67}, {4.2, 1.7},
   {3.89, 1.72}, {3.55, 1.84}, {3.53, 1.61}, {3.52, 1.33}, {3.51, 0.95}, {3.51, 0.48},
   {3.4, 0.25}, {3.11, 0.21}, {3.02, 0.34}, {3.17, 0.5}, {3.16, 0.63}, {3.07, 0.94}, {3., 1.27},
   {2.93, 1.48}, {2.81, 1.61}, {2.6, 1.4}, {2.37, 1.29}, {2.11, 1.1}, {1.83, 0.99}, {1.66, 0.7},
   {1.52, 0.46}, {1.33, 0.53}, {1.37, 0.76}, {1.47, 1.01}, {1.7, 1.3}, {1.91, 1.55}, {2.1, 1.73},
   {2.29, 1.92}, {2.54, 1.78}, {1.97, 2.17}, {1.76, 2.5}, {1.6, 2.74}, {1.36, 2.98}, {1.29, 2.92},
   {1.11, 2.88}, {0.87, 2.97}, {0.95, 3.16}, {0.95, 3.45}, {1.09, 3.78}, {1.31, 3.92},
   {1.46, 4.01}, {1.22, 4.11}, {1.09, 4.31}, {1.09, 4.39}, {1.3, 4.34}, {1.49, 4.23},
   {1.56, 4.07}, {1.95, 4.01}, {0.98, 3.91}, {0.88, 4.07}, {0.86, 4.26}, {1.04, 4.17},
   {5.99, 1.32}, {6.21, 1.08}, {6.52, 0.85}, {6.53, 0.61}, {6.53, 0.41}, {6.46, 0.26},
   {6.63, 0.16}, {6.84, 0.25}, {6.88, 0.47}, {6.89, 0.81}, {6.86, 1.12}, {6.69, 1.35},
   {6.56, 1.65}, {6.55, 1.94}, {6.6, 2.4}, {6.61, 3.39}, {6.77, 3.72}, {6.81, 4.18}, {6.76, 4.6},
   {6.7, 5.07}, {6.74, 5.51}, {6.99, 5.63}, {7.3, 5.71}, {7.41, 5.92}, {7.17, 6.11}, {6.74, 6.11},
   {6.34, 5.86}, {6.17, 5.46}, {6.08, 4.98}, {6.15, 4.57}, {6.25, 4.19}, {6.21, 3.77}};
```

In[1336]:=

```
IGLuneBetaSkeleton[points, #, PlotLabel → #] & /@ {0.8, 1, 2, 5}
```

Out[1336]=

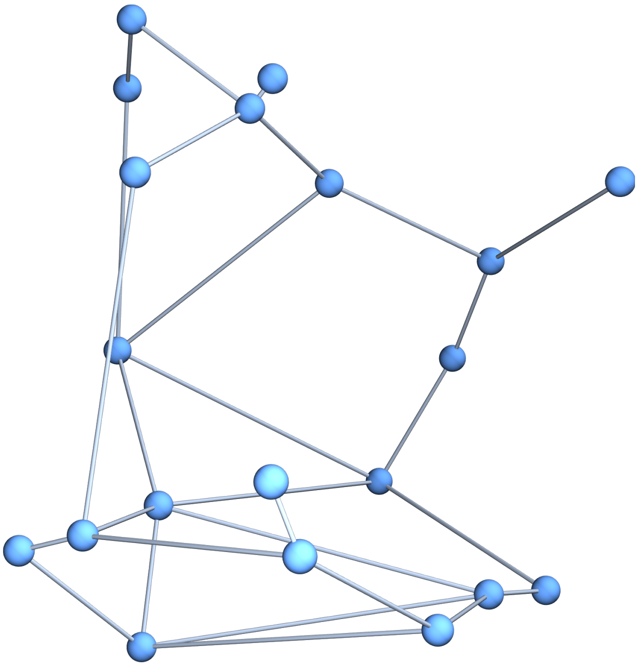



IGLuneBetaSkeleton works in 3D for $\beta \geq 1$.

In[1337]:=

```
IGLuneBetaSkeleton[RandomPoint[Ball[], 20], 1.5]
```

Out[1337]=



In[1338]:=

```
IGLuneBetaSkeleton[RandomPoint[Ball[], 20], 0.5]
```

IGraphM: Beta skeleton computation is only supported in 2 dimensions for beta < 1 or circle–based beta skeletons.

Out[1338]=

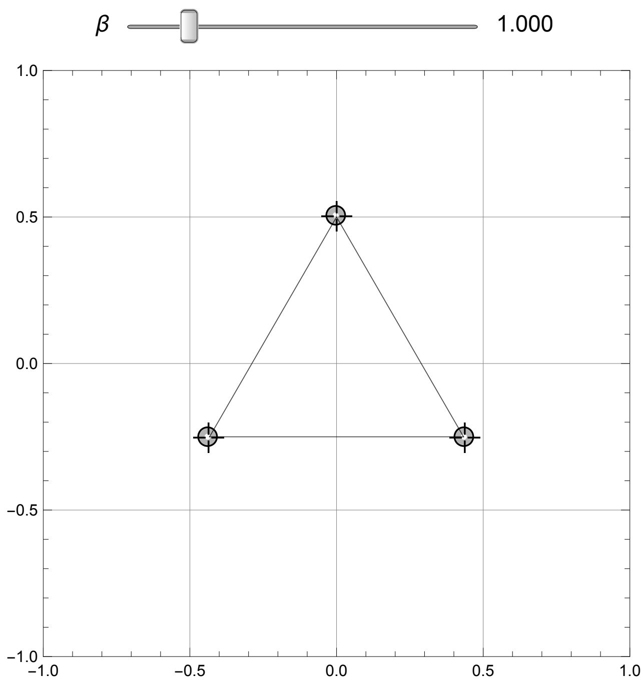
```
$Failed
```

Create a $\beta$-skeleton interactively. Point can be dragged around. To create or delete points, use CMD-click on macOS, ALT-click on Windows or CTRL-ALT-click on Linux.

In[1339]:=

```
DynamicModule[{pt = CirclePoints[0.5, 3], beta = 1.0},
 Grid[List /@ {
    Row[{Text@HoldForm[β], Spacer[10], Slider[Dynamic[beta], {1 / 2, 4}],
      Spacer[10], Text@Dynamic@NumberForm[beta, {3, 3}]}],
    LocatorPane[
     Dynamic[pt],
     Dynamic@IGLuneBetaSkeleton[pt, beta,
       PlotRange → {{-1, 1}, {-1, 1}},
       PlotRangePadding → 0,
       Frame → True,
       FrameTicks → Automatic,
       GridLines → Automatic,
       GraphStyle → "BasicBlack",
       ImageSize → Medium, VertexShape → None, VertexSize → 0
      ],
     LocatorAutoCreate → True
    ]
   }],
 Initialization :→ Needs["IGraphM`"]
]
```

Out[1339]=




## IGCircleBetaSkeleton

In[1340]:=

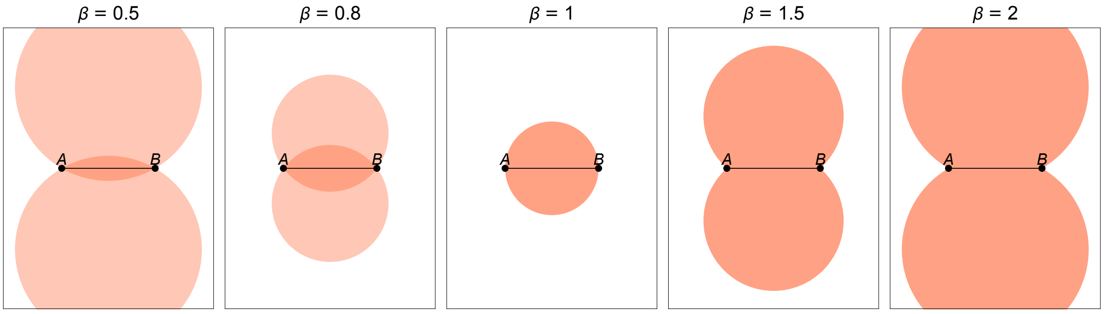
```
? IGCircleBetaSkeleton
```

IGCircleBetaSkeleton[points, beta] gives the circle–based beta skeleton of the given points.

The circle-based $\beta$ skeleton connects two points $A$ and $B$ if there is no other point $C$ so that the angle $\measuredangle ACB$ is less sharp the threshold

$$\theta = \begin{cases} \sin^{-1}\left(\frac{1}{\beta}\right) & \beta \geq 1 \\ \pi - \sin^{-1}(\beta) & \beta \leq 1 \end{cases}$$

This is equivalent to no point $C$ being contained in the intersection or unions of the disks illustrated below.



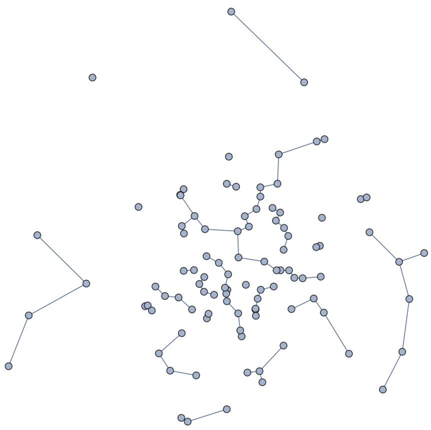

For $\beta \leq 1$, the circle based and lune based beta skeletons coincide. For $\beta > 1$, the circle-based beta skeleton is a subgraph of the lune-based one.

Compute the circle-based $\beta$ skeleton of a random point set.

In[1341]:=

```
IGCircleBetaSkeleton[RandomVariate[NormalDistribution[0, 1], {100, 2}], 1.2]
```

Out[1341]=



## IGRelativeNeighborhoodGraph

In[1342]:=

```
? IGRelativeNeighborhoodGraph
```

IGRelativeNeighborhoodGraph[points] gives the relative neighbourhood graph of the given points.

The relative neighbourhood graph is constructed from a set of points in space. Two points $A$ and $B$ are connected if and only if there is no other point $C$ so that $AC < AB$ and $BC < AB$, with the inequialities being strict.

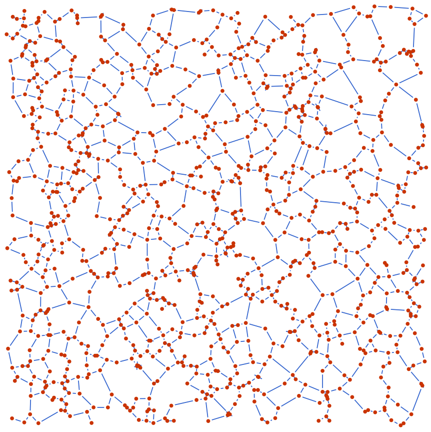
Most authors define the neighbourhood graph to coincide with a $\beta$-skeleton for $\beta = 2$. In IGraph/M, there is a subtle difference: the $\beta = 2$ skeleton connects points $A$ and $B$ when there is no point $C$ so that $AC \leq AB$ and $BC \leq AB$. Therefore, three points forming an equilateral triangle are connected in the relative neighborhood graph, but disconnected in the $\beta = 2$ skeleton.

Compute the relative neighbourhood graph of a random set of points.

In[1343]:=

```
g = IGRelativeNeighborhoodGraph[RandomReal[1, {1000, 2}],
  GraphStyle → "VibrantColor", VertexSize → {"Scaled", 0.005}]
```

Out[1343]=



Assign edge lengths as weights ...

In[1344]:=

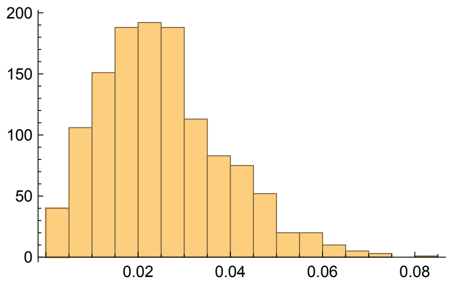
```
g = IGDistanceWeighted[g];
```

... and compute their distribution.

In[1345]:=

```
Histogram[IGEdgeProp[EdgeWeight][g]]
```

Out[1345]=




Plot the relative neighbourhood graph of African capitals.

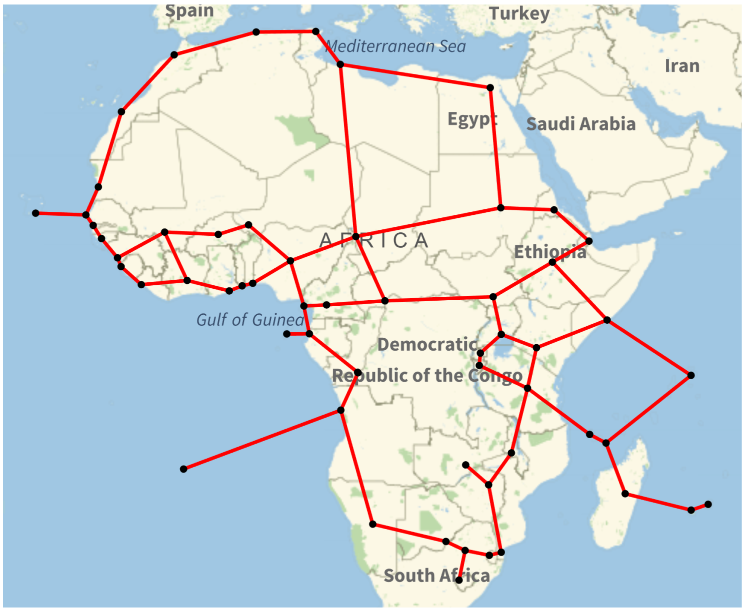
In[1346]:=

```
capitals = CountryData["Africa", "CapitalCity"];
edges = IGIndexEdgeList@IGRelativeNeighborhoodGraph@EntityValue[capitals, "Coordinates"];
```

In[1348]:=

```
GeoGraphics[{{Red, Thick, Line[capitals[[#]] & /@ edges]}, {PointSize[0.01], Point[capitals]}}]
```

Out[1348]=




When the point set contain equilateral triangles, the relative neighbourhood graph may not coincide with the $\beta = 2$ skeleton.

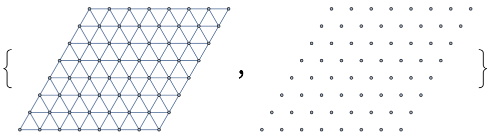
In[1349]:=

```
pts = MeshCoordinates@IGLatticeMesh["Triangular"];
```

In[1350]:=

```
{IGRelativeNeighborhoodGraph[pts], IGLuneBetaSkeleton[pts, 2]}
```

Out[1350]=



## IGGabrielGraph

In[1351]:=

```
? IGGabrielGraph
```

IGGabrielGraph[points] gives the Gabriel graph of the given points.

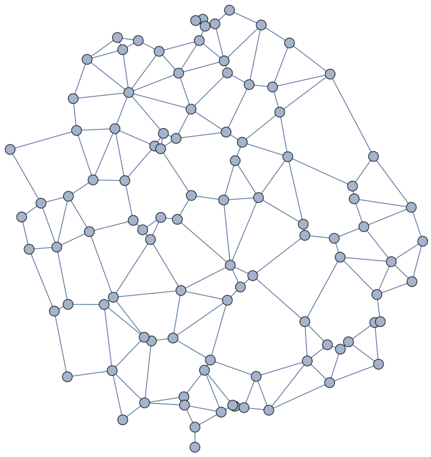
The Gabriel graph is constructed from a set of points in space. Two points $A$ and $B$ are connected if and only if no other point is contained in the disk of which $A B$ is a diameter.

The Gabriel graph coincides with a $\beta$-skeleton for $\beta = 1$.

In[1352]:=

```
pts = RandomPoint[Disk[], 100];
g = IGGabrielGraph[pts]
```

Out[1353]=

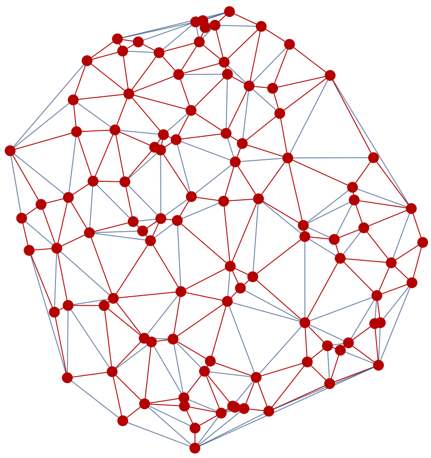


The Gabriel graph is a subgraph of the Delaunay graph.

In[1354]:=

```
HighlightGraph[IGMeshGraph@DelaunayMesh[pts], g]
```

Out[1354]=



Convert the Gabriel graph to a MeshRegion object by finding its faces, and removing the outer face. Here we use the heuristic that for a graph generated from a random point set, the face with the most vertices is likely to be the outer face.

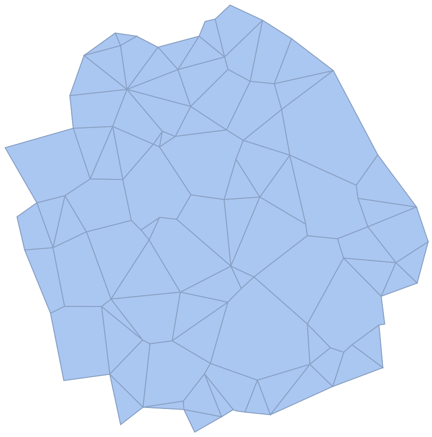
In[1355]:=

```
faces = IGFaces@IGCoordinatesToEmbedding[g];
faces = Delete[faces, Ordering[Length /@ faces, -1]];
```

In[1357]:=

```
MeshRegion[pts, Polygon[faces]]
```

Out[1357]=

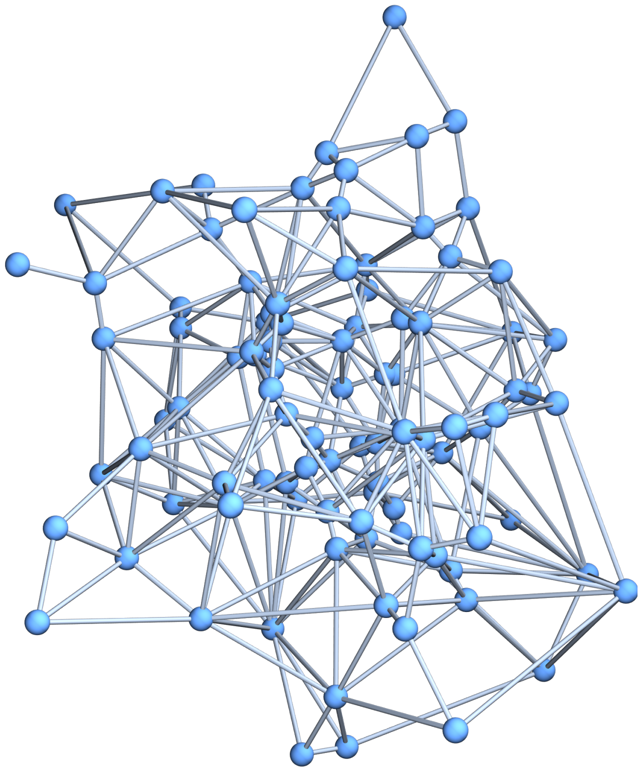


Compute a Gabriel graph in 3D.

In[1358]:=

```
IGGabrielGraph@RandomVariate[MultinormalDistribution@IdentityMatrix[3], 100]
```

Out[1358]=



## IGBetaWeightedGabrielGraph

In[1359]:=

```
? IGBetaWeightedGabrielGraph
```

IGBetaWeightedGabrielGraph[points] gives a Gabriel graph of points with edge weights representing $\beta$ values where the corresponding edge would disappear from a lune–based $\beta$–skeleton.

**Experimental:** This is experimental functionality that may change in the future.

`IGBetaWeightedGabrielGraph[points]` produces a Gabriel graph in which edge weights represent threshold $\beta$

values for lune-based $\beta$-skeletons. Each edge is present in $\beta$-skeletons having a $\beta$ parameter smaller than the threshold stored in its weight.

Available options:

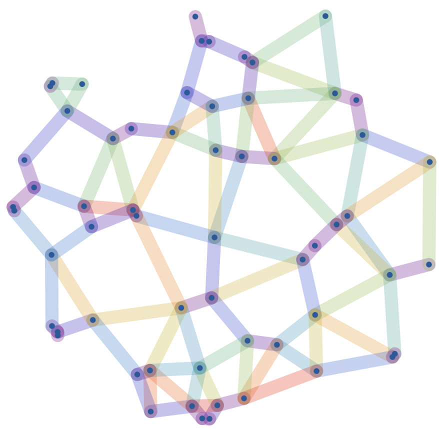
- `"BetaCutoff" → cutoff` only computes threshold $\beta$ values up `cutoff`. Larger thresholds will be returned as `Infinity`. This option is intended to increase performance: the lower the cutoff, the faster the computation. The default value is `Infinity`, i.e. no cutoff.

Colour edges by their inverse threshold $\beta$ values:

In[1360]:=
```
pts = RandomPoint[Disk[], 60];
```

In[1361]:=
```
g = IGBetaWeightedGabrielGraph[pts, GraphStyle → "ThickEdge"] //
  IGEdgeMap[ColorData["Rainbow"][1 / #] &, EdgeStyle → IGEdgeProp[EdgeWeight]]
```

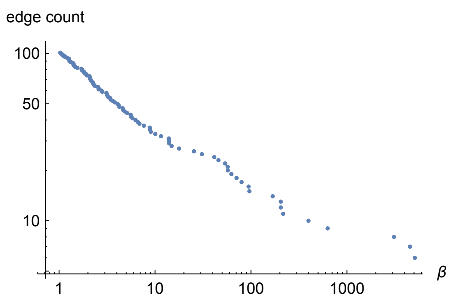
Out[1361]=



Plot the edge count of a lune-based $\beta$-skeleton of the same points as a function of $\beta$:

In[1362]:=
```
thresholds = IGEdgeProp[EdgeWeight][g];
ListLogLogPlot[
 Transpose[{Reverse@Sort[thresholds], Range@Length[thresholds]}],
 AxesLabel → {"β", "edge count"}
]
```

Out[1363]=

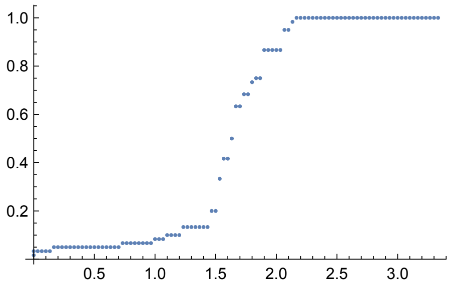


Show a percolation curve corresponding to edge removal in order of reverse threshold $\beta$:

In[1364]:=

```
ListPlot@IGPercolationCurve@Reverse@SortBy[EdgeList[g], PropertyValue[{g, #}, EdgeWeight] &]
```

Out[1364]=




# Weighted graphs

These functions perform basic operations on edge-weighted graphs without discarding edge weights. They are not based on the igraph C library.

## IGWeightedAdjacencyGraph

`IGWeightedAdjacencyGraph` constructs an edge-weighted graph from a weighted adjacency matrix. By default, `0` elements in the matrix are taken to indicate a lack of connection. An alternative value may be specified to indicate the lack of connections.

`IGWeightedAdjacencyGraph` takes the same options as `WeightedAjdacencyGraph`.

In[1365]:=

```
? IGWeightedAdjacencyGraph
```

IGWeightedAdjacencyGraph[matrix] creates a graph from a weighted adjacency matrix, taking 0 to mean unconnected.
IGWeightedAdjacencyGraph[vertices, matrix] uses vertices as the vertex names.
IGWeightedAdjacencyGraph[matrix, z] creates a graph
from a weighted adjacency matrix, taking the value z to mean unconnected.
IGWeightedAdjacencyGraph[vertices, matrix, z] uses vertices as the vertex names.

In[1366]:=

```
IGWeightedAdjacencyGraph[{{0, 2, 0}, {2, 0, 3}, {0, 3, 0}}, GraphStyle → "DiagramGold", EdgeLabels → "EdgeWeight"]
```

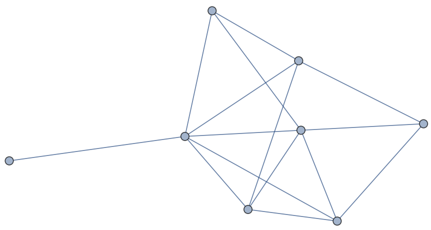
Out[1366]=

The built-in `WeightedAdjacencyGraph` uses `Infinity` to indicate the lack of connection. This makes it inconsistent with `WeightedAdjacencyMatrix`, which uses `0`. The purpose of `IGWeightedAdjacencyGraph` is to be able to easily interoperate with `WeightedAdjacencyMatrix`, and to easily cycle a weighted graph through an adjacency matrix representation.

In[1367]:=

```
wg = RandomGraph[{8, 15}, EdgeWeight → {_ :→ RandomReal[]}]
```

Out[1367]=



In[1368]:=

```
WeightedAdjacencyMatrix[wg] // MatrixForm
```

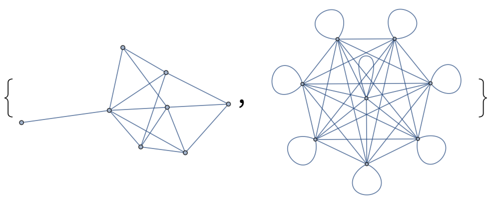
Out[1368]//MatrixForm=

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0. | 0. | 0.657029 | 0. | 0.619472 | 0. | 0.77406 | 0.669673 |
| 0. | 0. | 0. | 0. | 0.657961 | 0. | 0. | 0. |
| 0.657029 | 0. | 0. | 0.454061 | 0.149099 | 0.640649 | 0.326244 | 0. |
| 0. | 0. | 0.454061 | 0. | 0. | 0. | 0.565047 | 0.31078 |
| 0.619472 | 0.657961 | 0.149099 | 0. | 0. | 0.0491909 | 0.536172 | 0.268517 |
| 0. | 0. | 0.640649 | 0. | 0.0491909 | 0. | 0. | 0.431543 |
| 0.77406 | 0. | 0.326244 | 0.565047 | 0.536172 | 0. | 0. | 0. |
| 0.669673 | 0. | 0. | 0.31078 | 0.268517 | 0.431543 | 0. | 0. |

In[1369]:=

```
{IGWeightedAdjacencyGraph@WeightedAdjacencyMatrix[wg],
 WeightedAdjacencyGraph@WeightedAdjacencyMatrix[wg]}
```

Out[1369]=



In[1370]:=

```
PropertyValue[%, EdgeWeight]
```

Out[1370]=

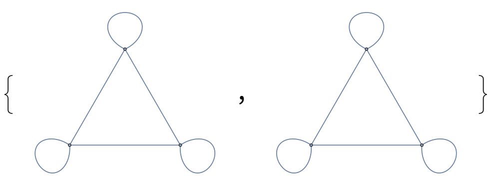
```
$Failed
```

`IGWeightedAdjacencyGraph[g, Infinity]` is equivalent to `WeightedAdjacencyGraph[g]`.

In[1371]:=

$$\{\texttt{IGWeightedAdjacencyGraph}\left[\begin{pmatrix} 0 & 2 & 0 \\ 2 & 0 & 3 \\ 0 & 3 & 0 \end{pmatrix}, \texttt{Infinity}\right], \texttt{WeightedAdjacencyGraph}\left[\begin{pmatrix} 0 & 2 & 0 \\ 2 & 0 & 3 \\ 0 & 3 & 0 \end{pmatrix}\right]\}$$

Out[1371]=

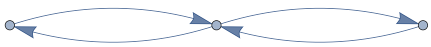


Use `DirectedEdges → True` to force creating a directed graph even from a symmetric matrix.

In[1372]:=

$$\texttt{IGWeightedAdjacencyGraph}\left[\begin{pmatrix} 0 & 2 & 0 \\ 2 & 0 & 3 \\ 0 & 3 & 0 \end{pmatrix}, \texttt{DirectedEdges} \to \texttt{True}\right]$$

Out[1372]=



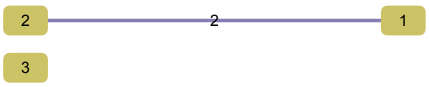
When the input matrix is not symmetric, `DirectedEdges → False` will cause the below-diagonal part of the matrix to be ignored.

In[1373]:=

$$\texttt{IGWeightedAdjacencyGraph}\left[\begin{pmatrix} 0 & 2 & 0 \\ 4 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}, \texttt{DirectedEdges} \to \texttt{False},\right.$$

```
 GraphStyle → "DiagramGold", EdgeLabels → "EdgeWeight"]
```

Out[1373]=




## IGWeightedAdjacencyMatrix

In[1374]:=

```
? IGWeightedAdjacencyMatrix
```

IGWeightedAdjacencyMatrix[graph] gives the adjacency matrix of the edge weights of graph.
IGWeightedAdjacencyMatrix[graph, z] gives the adjacency
matrix of the edge weights of graph, using the value z to represent absent connections.

`IGWeightedAdjacencyMatrix` is equivalent to the built-in `WeightedAdjacencyMatrix` with the difference that it allows specifying the value to use for representing absent connections.

By default, absent connections are represented with `0`. This does not allow for distinguishing between absent connections and connections with weight `0`.

In[1375]:=

```
g = Graph[{1 ⟷ 2, 2 ⟷ 3}, EdgeWeight → {2, 0}]
```

Out[1375]=



In[1376]:=

```
IGWeightedAdjacencyMatrix[g] // MatrixForm
```

Out[1376]//MatrixForm=

$$\begin{pmatrix} 0 & 2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

In[1377]:=

```
IGWeightedAdjacencyMatrix[g, Infinity] // MatrixForm
```

Out[1377]//MatrixForm=

$$\begin{pmatrix} \infty & 2 & \infty \\ 2 & \infty & 0 \\ \infty & 0 & \infty \end{pmatrix}$$

## IGEdgeWeightedQ

In[1378]:=
```
? IGEdgeWeightedQ
```

IGEdgeWeightedQ[graph] tests if graph is an edge-weighted graph.

Unlike `WeightedGraphQ`, `IGEdgeWeightedQ` does not return `True` for vertex-weighted graphs that have no edge weights.

In[1379]:=
```
g = Graph[{1 ↔ 2}, VertexWeight → {1.2, 2.3}];
```

In[1380]:=
```
IGEdgeWeightedQ[g]
```

Out[1380]=
```
False
```

In[1381]:=
```
IGEdgeWeightedQ[SetProperty[g, EdgeWeight → {1}]]
```

Out[1381]=
```
True
```

## IGVertexWeightedQ

In[1382]:=
```
? IGVertexWeightedQ
```

IGVertexWeightedQ[graph] tests if graph is a vertex-weighted graph.

Unlike `WeightedGraphQ`, `IGVertexWeightedQ` does not return `True` for edge-weighted graphs that have no vertex weights.

In[1383]:=
```
g = Graph[{1 ↔ 2}, EdgeWeight → {3.1}];
```

In[1384]:=
```
IGVertexWeightedQ[g]
```

Out[1384]=
```
False
```

In[1385]:=
```
IGVertexWeightedQ[SetProperty[g, VertexWeight → {2, 1}]]
```

Out[1385]=
```
True
```

## IGVertexStrength

`IGVertexStrength` returns the strength of each vertex, i.e. the total edge weight of edges adjacent to it.

In[1386]:=
```
? IGVertex*Strength
```

▼ IGraphM`

| | | |
|---|---|---|
| IGVertexInStrength | IGVertexOutStrength | IGVertexStrength |

In[1387]:=
```
g = ExampleData[{"NetworkGraph", "EurovisionVotes"}];
```

In[1388]:=

```
IGVertexStrength[g]
```

Out[1388]=

```
{25, 12, 40, 17, 25, 21, 45, 60, 16, 50, 38, 6, 62, 51, 40, 38, 11, 63, 74, 15, 44, 29, 45,
 48, 27, 29, 48, 24, 6, 18, 30, 63, 27, 27, 49, 83, 2, 43, 6, 28, 43, 57, 25, 91, 55, 40}
```

In- and out-strength can be calculated separately for directed graphs.

In[1389]:=

```
{IGVertexInStrength[g], IGVertexOutStrength[g]}
```

Out[1389]=

```
{{11, 0, 30, 3, 19, 7, 20, 38, 4, 24, 13, 0, 40, 25, 20, 12, 5, 37, 52, 3, 22, 5,
  21, 26, 7, 9, 22, 12, 0, 10, 7, 39, 5, 6, 27, 61, 0, 31, 0, 4, 17, 31, 5, 65, 39, 14},
 {14, 12, 10, 14, 6, 14, 25, 22, 12, 26, 25, 6, 22, 26, 20, 26, 6, 26, 22, 12, 22, 24, 24,
  22, 20, 20, 26, 12, 6, 8, 23, 24, 22, 21, 22, 22, 2, 12, 6, 24, 26, 26, 20, 26, 16, 26}}
```

In[1390]:=

```
IGVertexInStrength[g] + IGVertexOutStrength[g] == IGVertexStrength[g]
```

Out[1390]=

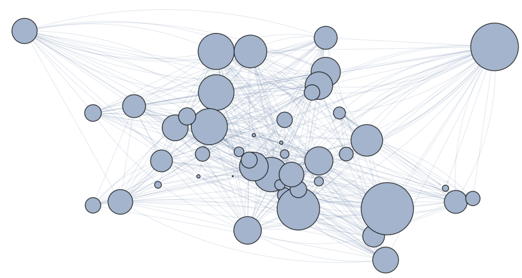
```
True
```

Scale vertices by strength:

In[1391]:=

```
IGVertexMap[{"Scaled", 0.1 #} &, VertexSize → (Normalize[IGVertexStrength[#], Max] &), g]
```

Out[1391]=



## IGUnweighted

`IGUnweighted[g]` returns a version of the graph g with edge weights removed.

In[1392]:=

```
? IGUnweighted
```

IGUnweighted[graph] returns an unweighted version of an edge–weighted graph, while preserving other graph properties.

This function is useful for computing graph properties such as betweenness centrality without taking weights into account.

In[1393]:=

```
g = ExampleData[{"NetworkGraph", "EastAfricaEmbassyAttacks"}];
```

In[1394]:=

```
IGBetweenness[g]
```

Out[1394]=

```
{2., 2., 1., 0., 8., 0., 8.5, 23.5, 0.5, 14.5, 7., 3.5, 71.5, 3., 7., 0., 0., 30.}
```

In[1395]:=

```
IGBetweenness@IGUnweighted[g]
```

Out[1395]=

```
{3.49167, 0.416667, 3.75, 0., 3.48333, 1.79167, 11.9167,
 32.3417, 7.38333, 5.9, 6.65, 1.36667, 20.275, 8.45, 23.4833, 0., 0., 2.3}
```

## IGDistanceWeighted

`IGDistanceWeighted[g]` returns a weighted version of the graph g, setting the weights of edges to the distance between their endpoints. The distances are computed based on the `VertexCoordinates` property.

In[1396]:=

```
? IGDistanceWeighted
```

IGDistanceWeighted[graph] sets the weight of each edge to be the geometrical distance between its endpoints.

The available options are:

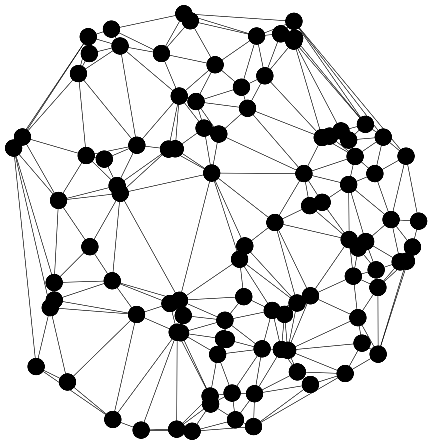
- `DistanceFunction` sets the function used to compute distances. The default is `EuclideanDistance`.

Create a Delaunay graph.

In[1397]:=

```
pts = RandomPoint[Disk[], 100];
g = IGDelaunayGraph[pts, GraphStyle → "BasicBlack", VertexSize → {"Scaled", 0.03}]
```

Out[1398]=



`IGDelaunayGraph` returns unweighted graphs.

In[1399]:=

```
IGEdgeWeightedQ[g]
```

Out[1399]=

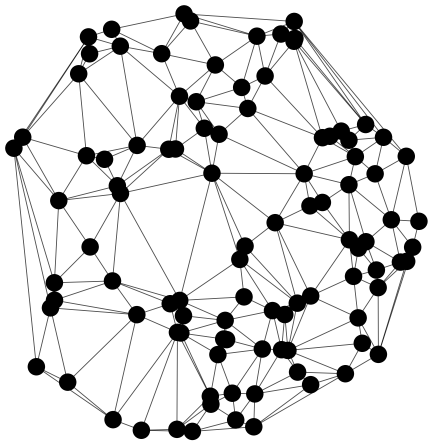
```
False
```

Set edge weights based on the geometrical distances between the edge endpoints.

In[1400]:=

```
wg = IGDistanceWeighted[g]
```

Out[1400]=



In[1401]:=

```
IGEdgeWeightedQ[wg]
```

Out[1401]=

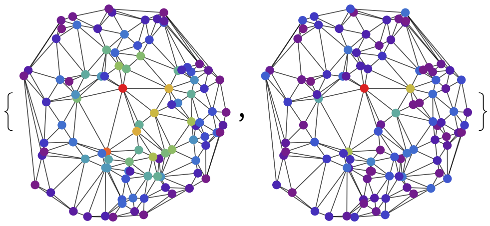
```
True
```

Edge weights are taken into account by many graph analysis functions.

In[1402]:=

```
IGVertexMap[ColorData["Rainbow"], VertexStyle → Rescale@*IGBetweenness] /@ {wg, g}
```

Out[1402]=



Compute the mean edge length.

In[1403]:=

```
Mean[IGEdgeProp[EdgeWeight][wg]]
```

Out[1403]=

```
0.218076
```

Use Manhattan distances instead of Euclidean distances.

In[1404]:=

```
IGDistanceWeighted[g, DistanceFunction → ManhattanDistance] // IGEdgeProp[EdgeWeight] // Mean
```

Out[1404]=

```
0.278232
```

`IGDistanceWeighted[g]` is effectively equivalent to the following `IGEdgeMap` construct, but faster for several specific distance functions:

In[1405]:=

```
wg1 = IGEdgeMap[Apply[EuclideanDistance], EdgeWeight → IGEdgeVertexProp[VertexCoordinates], g]; //
  RepeatedTiming
```

Out[1405]=

```
{0.000985902, Null}
```

In[1406]:=

```
wg2 = IGDistanceWeighted[g]; // RepeatedTiming
```

Out[1406]=

```
{0.000324609, Null}
```

In[1407]:=

```
IGEdgeProp[EdgeWeight][wg1] == IGEdgeProp[EdgeWeight][wg2]
```

Out[1407]=

```
True
```

## IGWeightedSimpleGraph

In[1408]:=

```
? IGWeightedSimpleGraph
```

IGWeightedSimpleGraph[graph] combines parallel edges by adding their weights. If graph is not weighted, the resulting weights will be the edge multiplicities of graph.
IGWeightedSimpleGraph[graph, comb] applies the function comb to the weights of parallel edges to compute a new weight. The default combiner is Plus.

`IGWeightedSimpleGraph` creates an edge-weighted graph by combining the weights of parallel edges. The default combiner function is `Plus`. If the input is an unweighted graph, the resulting weights will be the edge multiplicities of the input graph.

Available options:

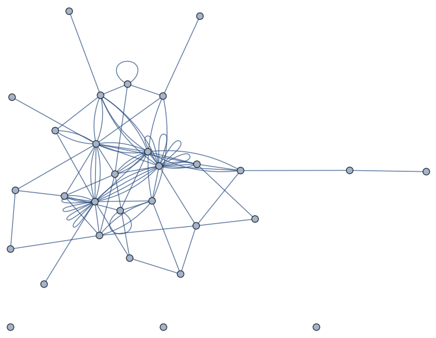
- `SelfLoops → False` will discard self-loops. The default is to keep them.

Convert edge multiplicities to weights.

In[1409]:=

```
g = IGGrowingGame[30, 3]
```

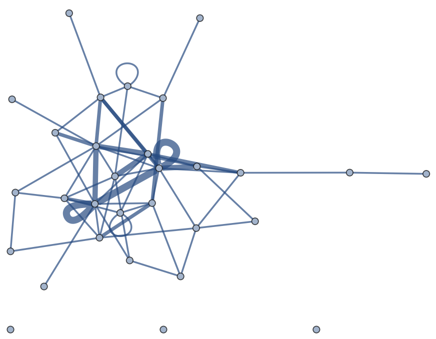
Out[1409]=



In[1410]:=

```
IGWeightedSimpleGraph[g] // IGEdgeMap[AbsoluteThickness, EdgeStyle → IGEdgeProp[EdgeWeight]]
```

Out[1410]=

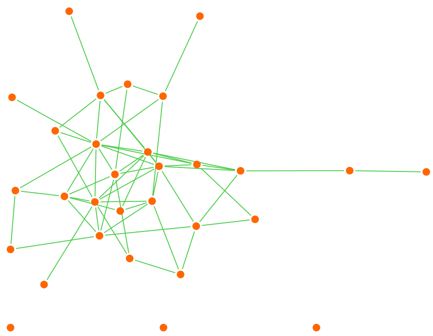


Discard self-loops and apply additional graph options.

In[1411]:=

```
IGWeightedSimpleGraph[g, SelfLoops → False, PlotTheme → "NeonColor"]
```

Out[1411]=



In[1412]:=

```
IGEdgeWeightedQ[%]
```

Out[1412]=

```
True
```

Combine edge weights by adding, multiplying or averaging, as controlled by the second argument of `IGWeightedSimpleGraph`.

In[1413]:=

```
MatrixForm@WeightedAdjacencyMatrix@IGWeightedSimpleGraph[
      Graph[{1 ⟷ 2, 1 ⟷ 2, 2 ⟷ 3}, EdgeWeight → {2, 3, 4}],
      #
    ] & /@ {Plus, Times, Mean[{##}] &}
```

Out[1413]=

$$\left\{\begin{pmatrix} 0 & 5 & 0 \\ 5 & 0 & 4 \\ 0 & 4 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 6 & 0 \\ 6 & 0 & 4 \\ 0 & 4 & 0 \end{pmatrix}, \begin{pmatrix} 0 & \frac{5}{2} & 0 \\ \frac{5}{2} & 0 & 4 \\ 0 & 4 & 0 \end{pmatrix}\right\}$$

## IGWeightedUndirectedGraph

In[1414]:=

```
? IGWeightedUndirectedGraph
```

IGWeightedUndirectedGraph[graph] converts an edge-weighted
directed graph to an undirected one. The weights of reciprocal edges added up.
IGWeightedUndirectedGraph[graph, comb] applies the function comb to the
weights of reciprocal edges to compute the weight of the corresponding undirected edge.
IGWeightedUndirectedGraph[graph, None] converts each directed edge
to an undirected one without combining their weights. The result may be a multigraph.

`IGWeightedUndirectedGraph` works like the built-in `UndirectedGraph`, but preserves edge weights. The weights of reciprocal edges will be combined with the given combiner function. By default, `Plus` is used, i.e. they are added up.

In[1415]:=

```
IGWeightedUndirectedGraph[
  Graph[{1 → 2, 2 → 1, 2 → 3}, EdgeWeight → {3, 4, 5}]
 ] // WeightedAdjacencyMatrix // MatrixForm
```

Out[1415]//MatrixForm=

$$\begin{pmatrix} 0 & 7 & 0 \\ 7 & 0 & 5 \\ 0 & 5 & 0 \end{pmatrix}$$

Average weights instead of adding them.

In[1416]:=

```
IGWeightedUndirectedGraph[
  Graph[{1 → 2, 2 → 1, 2 → 3}, EdgeWeight → {3, 4, 5}],
  Mean[{##}] &
 ] // WeightedAdjacencyMatrix // MatrixForm
```

Out[1416]//MatrixForm=

$$\begin{pmatrix} 0 & \frac{7}{2} & 0 \\ \frac{7}{2} & 0 & 5 \\ 0 & 5 & 0 \end{pmatrix}$$

This function is not meant to be used with multigraphs. If the input is a multigraph, weights of parallel edges will be combined with the same combiner function that is used for reciprocal edges. This may lead to unexpected results, thus a warning is issued.

In[1417]:=

```
IGWeightedUndirectedGraph[
  Graph[{1 → 2, 1 → 2, 2 → 1, 2 → 3}, EdgeWeight → {3, 4, 4, 5}],
  Mean[{##}] &
 ] // WeightedAdjacencyMatrix // MatrixForm
```

IGWeightedUndirectedGraph : The input is a multigraph. Weights of parallel edges will be combined with the same combiner function as used for reciprocal edges.

Out[1417]//MatrixForm=

$$\begin{pmatrix} 0 & \frac{11}{3} & 0 \\ \frac{11}{3} & 0 & 5 \\ 0 & 5 & 0 \end{pmatrix}$$

Use `IGWeightedSimpleGraph` to combine parallel edges before converting the graph to undirected.

In[1418]:=
```
IGWeightedUndirectedGraph[
   IGWeightedSimpleGraph[
    Graph[{1 → 2, 1 → 2, 2 → 1, 2 → 3}, EdgeWeight → {3, 4, 4, 5}],
    Mean[{##}] &
   ],
   Mean[{##}] &
  ] // WeightedAdjacencyMatrix // MatrixForm
```

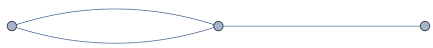
Out[1418]//MatrixForm=

$$\begin{pmatrix} 0 & \frac{15}{4} & 0 \\ \frac{15}{4} & 0 & 5 \\ 0 & 5 & 0 \end{pmatrix}$$

If None is used for the combiner, reciprocal edges are not combined. A weighted multigraph is created instead.

In[1419]:=
```
IGWeightedUndirectedGraph[
 Graph[{1 → 2, 2 → 1, 2 → 3}, EdgeWeight → {3, 4, 5}],
 None
]
```

Out[1419]=



In[1420]:=
```
IGEdgeWeightedQ[%]
```

Out[1420]=
```
True
```

## IGWeightedVertexDelete

In[1421]:=
```
? IGWeightedVertexDelete
```

IGWeightedVertexDelete[graph, vertex] deletes the given vertex while preserving edge weights.
IGWeightedVertexDelete[graph, {v1, v2, …}] deletes the given set of vertices while preserving edge weights.

In *Mathematica* 11.3 and earlier, the built-in `VertexDelete` does not handle edge weights correctly, and may sometimes produce `Graph` expressions with a broken internal structure. The purpose of `IGWeightedVertexDelete` is to provide a fast and reliable way to remove a vertex while preserving edge weights. Only edge weights are retained. All other properties are discarded.

In[1422]:=
```
g = Graph[{1 ↔ 2, 2 ↔ 3}, EdgeWeight → {4, 5}];
```

In[1423]:=
```
IGEdgeProp[EdgeWeight][g]
```

Out[1423]=
```
{4, 5}
```

In[1424]:=
```
IGEdgeProp[EdgeWeight]@IGWeightedVertexDelete[g, 3]
```

Out[1424]=
```
{4}
```

**Possible issues:**

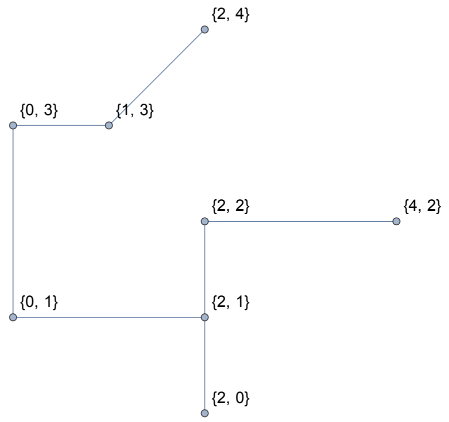
To delete a single vertex whose name is a list, it is necessary to use the syntax `IGWeightedVertexDelete[g, {v}]` to avoid ambiguity.

Graphs with list vertices commonly appear in the output of several functions, such as `NearestNeighborGraph`, `RelationGraph`, `IGDisjointUnion` or `IGMeshCellAdjacencyGraph`.

In[1425]:=

```
g = NearestNeighborGraph[RandomInteger[4, {10, 2}], VertexLabels → "Name"] //
  IGEdgeMap[Apply[EuclideanDistance], EdgeWeight → EdgeList]
```

Out[1425]=




In[1426]:=

```
v = First@VertexList[g]
```

Out[1426]=

```
{2, 2}
```

In this case, the single-vertex convenience syntax will not work.

In[1427]:=

```
IGWeightedVertexDelete[g, v]
```

IGraphM: The vertex 2 does not exist in the graph.

Out[1427]=

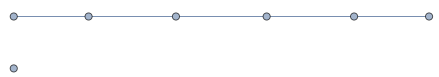
```
$Failed
```

Wrap the vertex in a list instead.

In[1428]:=

```
IGWeightedVertexDelete[g, {v}]
```

Out[1428]=



## IGWeightedSubgraph

In[1429]:=

```
? IGWeightedSubgraph
```

IGWeightedSubgraph[graph, {v1, v2, …}] returns the subgraph induced by the given vertices while preserving edge weights.

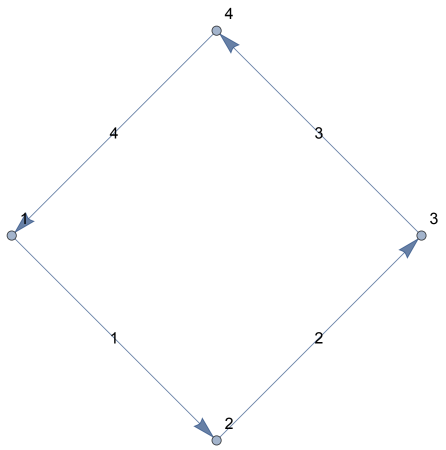
In *Mathematica* 11.3 and earlier, the built-in `Subgraph` function does not preserve edge weights. `IGWeightedSubgraph` preserves edge weights, but discards all other properties.

To retain not only edge weights, but also other properties, use `IGTakeSubgraph`. `IGWeightedSubgraph` offers much better performance than `IGTakeSubgraph` at the cost of discarding other properties.

In[1430]:=

```
g = Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1},
  EdgeWeight → {1, 2, 3, 4}, EdgeLabels → "EdgeWeight", VertexLabels → "Name"]
```

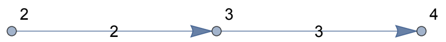
Out[1430]=




In[1431]:=

```
IGWeightedSubgraph[g, {2, 3, 4}, EdgeLabels → "EdgeWeight", VertexLabels → "Name"]
```

Out[1431]=




# Degree sequences

## Graphicality

A sequence of integers is called *graphical* if there is an undirected graph that has them as its degree sequence. Some authors apply the term *graphical* only when the degrees can be realized by a *simple* graph. Here we use it in a more general sense, as IGraph/M is able to perform the test also for the cases of multigraphs with loops, loop-free multigraphs and simple graph with at most one self-loop per vertex. These are controlled by the `SelfLoops` and `MultiEdges` options. The concept of graphicality generalizes to pairs of in- and out-degree sequences of directed graphs as well.

### IGGraphicalQ

In[1432]:=

```
? IGGraphicalQ
```

IGGraphicalQ[degrees] tests if degrees is the degree sequence of any simple undirected graph.
IGGraphicalQ[indegrees, outdegrees] tests if
  indegrees with outdegrees is the degree sequence of any simple directed graph.
IGGraphicalQ[degrees, SelfLoops -> True] tests if degrees is
  the degree sequence of any undirected graph with at most one self–loop per vertex.
IGGraphicalQ[degrees, MultiEdges -> True] tests if degrees is the degree sequence of any undirected loop–free multigraph.

In the undirected case, `IGGraphicalQ` uses the Erdős–Gallai theorem to check if the degree sequence is realized by any simple graph. For loopy multigraphs, it is sufficient to check that the sum of degrees is even. If self-loops are disallowed, there is the additional condition that $\sum_i d_i \geq d_{\max}$. If at most one self-loop is allowed per vertex, but no multi-edges, a modification of the Erdős-Gallai conditions due to Cairns and Mendan are used.

In the directed case, `IGGraphicalQ` uses the Fulkerson–Chen–Antsee theorem with Berger's refinement. For loopy multi-digraphs, it is sufficient to check that the sum of in-degrees equals the sum of out-degrees. If self-loops are disallowed, there is the additional condition that the sum of in-degrees (or, alternatively, the sum of out-degrees) is not smaller than the maximum total degree. If at most one self-loop is allowed per vertex, but no multi-edges, the problem becomes equivalent to realizability as a simple bipartite graph, and the Gale–Ryser theorem can be used (see `IGBigraphicalQ`).

To actually construct a realization, use the `IGRealizeDegreeSequence` function. To sample random realizations, use `IGDegreeSequenceGame`.

The allowed options are:

- `SelfLoops → True` checks if the degree sequence has realizations that potentially contain self-loops.
- `MultiEdges → True` checks if the degree sequence has realizations that potentially contain more than one connection between pairs of vertices.

Check if a degree sequence is graphical …

In[1433]:=

```
IGGraphicalQ[{4, 3, 3, 2, 1, 1}]
```

Out[1433]=

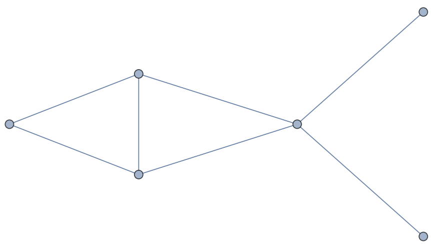
```
True
```

… then construct a realization as a simple graph:

In[1434]:=

```
IGRealizeDegreeSequence[{4, 3, 3, 2, 1, 1}]
```

Out[1434]=



Check the same for a pair of in- and out-degree sequences, the construct a realization as a simple directed graph:

In[1435]:=

```
{IGGraphicalQ[{0, 2, 0}, {1, 0, 1}], IGGraphicalQ[{1, 0, 1}, {0, 2, 0}]}
```

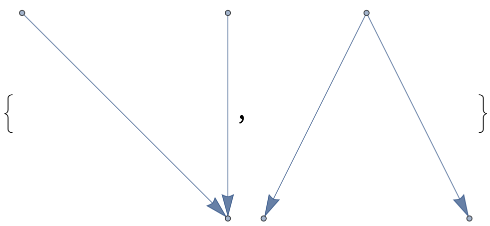
Out[1435]=

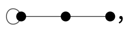
```
{True, True}
```

In[1436]:=

```
{IGRealizeDegreeSequence[{0, 2, 0}, {1, 0, 1}], IGRealizeDegreeSequence[{1, 0, 1}, {0, 2, 0}]}
```

Out[1436]=



The degree sequence (1, 2, 3) has no realization as a simple graph, but it can be realized either as a simple loopy graph, , or as a loop-free multigraph, .

In[1437]:=

```
{IGGraphicalQ[{1, 2, 3}],
 IGGraphicalQ[{1, 2, 3}, SelfLoops → True],
 IGGraphicalQ[{1, 2, 3}, MultiEdges → True]}
```

Out[1437]=

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

(4, 1, 1) is realizable as a loopy simple graph, but not as a loop-free multigraph.

In[1438]:=

```
{IGGraphicalQ[{4, 1, 1}, SelfLoops → True],
 IGGraphicalQ[{4, 1, 1}, MultiEdges → True]}
```

Out[1438]=

```
{True, False}
```

Any graph with the degree sequence (6, 2, 2) must have both self-loops and multi-edges.

In[1439]:=

```
TableForm[
 Outer[
  IGGraphicalQ[{6, 2, 2}, SelfLoops → #1, MultiEdges → #2] &,
  {False, True}, {False, True}
 ], TableHeadings → {{"self-loops", "multi-edges"}, {"self-loops", "multi-edges"}}]
```

Out[1439]//TableForm=

| | self-loops | multi-edges |
|---|---|---|
| self-loops | False | False |
| multi-edges | False | True |

The following pair of in- and out-degree sequences can be realized as a directed graph with at most one self-loop per vertex, but not as a loop-free multigraph:

In[1440]:=

```
{IGGraphicalQ[{1, 0, 2}, {0, 1, 2}, SelfLoops → True],
 IGGraphicalQ[{1, 0, 2}, {0, 1, 2}, MultiEdges → True]}
```

Out[1440]=

```
{True, False}
```

Create a random graphical scale-free degree sequence and construct a corresponding graph:

In[1441]:=

```
ds = IGTryUntil[IGGraphicalQ]@RandomVariate[ZipfDistribution[1.1], 100]
```

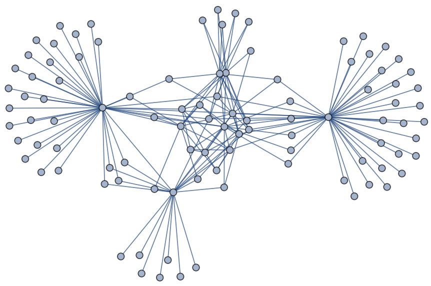
Out[1441]=

```
{2, 14, 2, 1, 1, 2, 7, 1, 1, 1, 1, 1, 1, 11, 1, 1, 2, 1, 1, 3, 5, 1, 1, 1, 2, 9, 1, 1, 2, 7, 1, 1, 1, 1, 1,
 5, 1, 1, 2, 2, 1, 1, 41, 2, 1, 3, 1, 1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 3, 2, 2, 1, 1, 1, 1, 4, 39, 1,
 1, 4, 2, 4, 1, 1, 1, 3, 1, 1, 1, 1, 1, 14, 2, 2, 1, 1, 1, 1, 2, 10, 1, 1, 1, 4, 4, 1, 14, 2, 21, 7}
```

In[1442]:=

```
IGRealizeDegreeSequence[ds]
```

Out[1442]=



### References


- P. Erdős and T. Gallai, Gráfok Előírt Fokú Pontokkal, Matematikai Lapok 11, 264 (1960). https://users.renyi.hu/~p_erdos/1961-05.pdf
- G. Cairns and S. Mendan, Degree Sequences for Graphs with Loops, 1 (2013). https://arxiv.org/abs/1303.2145v1
- Z. Király, Recognizing Graphic Degree Sequences and Generating All Realizations, No. TR–2011–11, Egerváry Research Group, Eötvös Loránd University, 2012. http://bolyai.cs.elte.hu/egres/tr/egres-11-11.pdf
- B. Cloteaux, Is This for Real? Fast Graphicality Testing, Computing in Science & Engineering 17, 6 (2015). https://dx.doi.org/10.1109/MCSE.2015.125
- D. R. Fulkerson, Zero-One Matrices with Zero Trace, Pacific Journal of Mathematics 10, 3 (1960). https://doi.org/10.2140/pjm.1960.10.831
- W. K. Chen, On the Realization of a (*p*, *s*)-digraph with Prescribed Degrees, Journal of the Franklin Institute 281, 5 (1966). https://doi.org/10.1016/0016-0032(66)90301-2
- R. P. Anstee, Properties of a Class of (0, 1)-matrices Covering a Given Matrix, Canadian Journal of Mathematics 34, 2 (1982). https://doi.org/10.4153/cjm-1982-029-3
- A. Berger, A Note on the Characterization of Digraphic Sequences, Discrete Mathematics 314, 1 (2014). https://dx.doi.org/10.1016/j.disc.2013.09.010
- Sz. Horvát and C. D. Modes, Connectivity matters: Construction and exact random sampling of connected graphs (2020). https://arxiv.org/abs/2009.03747


## IGBigraphicalQ

In[1443]:=

```
? IGBigraphicalQ
```

IGBigraphicalQ[degrees1, degrees2] tests if (degrees1, degrees2) is the degree sequence of any bipartite simple graph.
IGBigraphicalQ[degrees1, degrees2, MultiEdges -> True]
tests if (degrees1, degrees2) is the degree sequence of any bipartite multigraph.

`IGBigraphicalQ[degrees1, degrees2]` checks if there is a bipartite graph which has `degrees1` and `degrees2` as the vertex degrees in the two partitions. Such a pair of degree sequences is called bigraphical.

If multi-edges are allowed in the graph, it is sufficient to check that the two degree sequences sum to the same value. If only simple graphs are allowed, `IGBigraphicalQ` uses the Gale–Ryser theorem with Berger's refinement.

The available options are:

- `MultiEdges` → `True` allows for multi-edges in the graph.

In[1444]:=
```
IGBigraphicalQ[{4, 3, 1, 2, 4}, {2, 4, 4, 4}]
```
Out[1444]=
```
True
```

The following pair of degree sequences is bigraphical only if multi-edges are permitted:

In[1445]:=
```
{IGBigraphicalQ[{2, 2}, {4}, MultiEdges → False], IGBigraphicalQ[{2, 2}, {4}, MultiEdges → True]}
```
Out[1445]=
```
{False, True}
```

#### References


- H. J. Ryser, Combinatorial Properties of Matrices of Zeros and Ones, Can. J. Math. 9, 371 (1957). https://dx.doi.org/10.4153/cjm-1957-044-3
- D. Gale, A theorem on flows in networks, Pacific J. Math. 7, 1073 (1957). https://dx.doi.org/10.2140/pjm.1957.7.1073
- A. Berger, A Note on the Characterization of Digraphic Sequences, Discrete Mathematics 314, 1 (2014). https://dx.doi.org/10.1016/j.disc.2013.09.010


## Potential connectedness

### IGPotentiallyConnectedQ

In[1446]:=
```
? IGPotentiallyConnectedQ
```

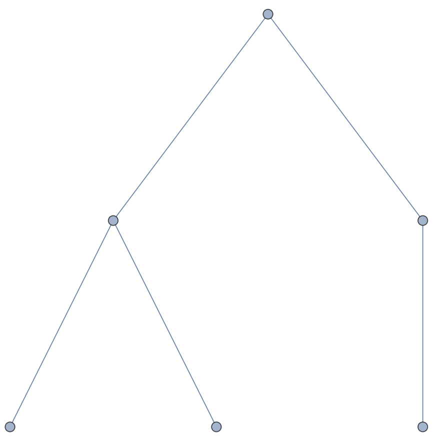
IGPotentiallyConnectedQ[degrees] tests if degrees is the degree sequence of some connected graph.

`IGPotentiallyConnectedQ` checks if a degree sequence has a realization as a connected graph. The condition is that the degree sequence $(d_1, \ldots, d_n)$ satisfy $\frac{1}{2} \sum_i d_i \geq n - 1$ and $d_i > 0$ for all $i$. Additionally, `IGPotentiallyConnectedQ` requires that the sum of degrees be odd.

Check the potential connectivity of a degree sequence, then construct a corresponding connected graph:

In[1447]:=
```
IGPotentiallyConnectedQ[{3, 2, 2, 1, 1, 1}]
```
Out[1447]=
```
True
```

In[1448]:=
```
IGRealizeDegreeSequence[{3, 2, 2, 1, 1, 1}, Method → "SmallestFirst"]
```
Out[1448]=



The empty degree sequence is considered non-potentially-connected:

In[1449]:=
```
IGPotentiallyConnectedQ[{}]
```
Out[1449]=
```
False
```

The length-1 degree sequence (0) is considered connected:

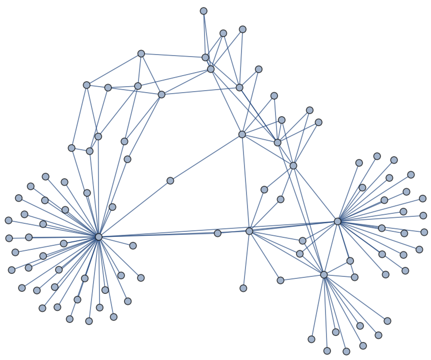
In[1450]:=
```
IGPotentiallyConnectedQ[{0}]
```
Out[1450]=
```
True
```

`IGPotentiallyConnectedQ` returns `False` for odd-sum sequences, as no graph can have them as its degrees:

In[1451]:=
```
IGPotentiallyConnectedQ[{3, 3, 3}]
```
Out[1451]=
```
False
```

Generate a random potentially connected degree sequence and construct a corresponding connected non-simple graph:

In[1452]:=
```
IGRealizeDegreeSequence[
 IGTryUntil[IGPotentiallyConnectedQ]@RandomVariate[ZipfDistribution[1], 100],
 MultiEdges → True, SelfLoops → True, Method → "SmallestFirst"
]
```
Out[1452]=



## Constructing graphs

See the Graph creation section for a detailed description of these functions.

In[1453]:=
```
? IGRealizeDegreeSequence
```

IGRealizeDegreeSequence[degrees] gives an undirected graph having the given degree sequence. Available Method options: {"SmallestFirst", "LargestFirst", "Index"}.
IGRealizeDegreeSequence[indegrees, outdegrees] gives a directed graph having the given out– and in–degree sequences.

In[1454]:=
```
? IGDegreeSequenceGame
```

IGDegreeSequenceGame[degrees] generates an undirected random graph with the given degree sequence. Available Method options: {"ConfigurationModel", "ConfigurationModelSimple", "FastSimple", "VigerLatapy"}.
IGDegreeSequenceGame[indegrees, outdegrees] generates a directed random graph with the given in– and out–degree sequences.

# Threshold graphs

## IGSplitQ

In[1455]:=

```
? IGSplitQ
```

IGSplitQ[graph] tests if graph is a split graph.
IGSplitQ[degrees] tests if degrees is the degree sequence of a split graph.

`IGSplitQ` recognizes split graphs, or their degree sequences. Splits graphs are the graphs which can be partitioned into a clique and an independent set. `IGSplitQ` ignores self-loops and multi-edges.

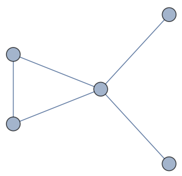
Split graph can be recognized solely based on their degree sequence. Let $d_1 \geq d_2 \geq \cdots \geq d_n$ be the non-increasingly ordered degree sequence, and $m$ the largest index $i$ such that $d_i \geq i-1$. Then the graph is split if and only if $\sum_{i=1}^{m} d_i = m(m-1) + \sum_{i=m+1}^{n} d_i$.

When applied to a list of integers, `IGSplitQ` only checks this condition, but does not verify graphicality.

Test if a graph is split:

In[1456]:=

```
g = 
```



```
;
```

In[1457]:=

```
IGSplitQ[g]
```

Out[1457]=

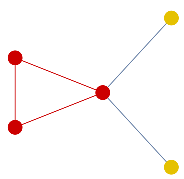
```
True
```

Highlight its clique and independent vertex set parts:

In[1458]:=

```
cl = First@IGLargestCliques[g];
HighlightGraph[g, Subgraph[g, #] & /@ {cl, Complement[VertexList[g], cl]}]
```

Out[1459]=



`IGSplitQ` can be used directly on degree sequences:

In[1460]:=

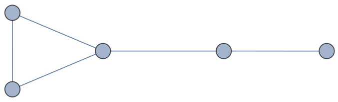
```
IGSplitQ@VertexDegree[g]
```

Out[1460]=

```
True
```

The following graph is not split:

In[1461]:=

```
IGSplitQ[
```

```
]
```

Out[1461]=

```
False
```

## IGThresholdQ

In[1462]:=

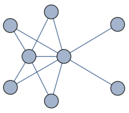
```
? IGThresholdQ
```

IGThresholdQ[graph] tests if graph is a threshold graph.
IGThresholdQ[degrees] tests if degrees form a threshold degree sequence.

`IGThresholdQ` recognizes threshold graphs, or their degree sequences. `IGThresholdQ` ignores self-loops and multi-edges.

Check if the following is a threshold graph:

In[1463]:=

```
g = 
```

```
;
```

In[1464]:=

```
IGThresholdQ[g]
```

Out[1464]=

```
True
```

Threshold graphs are also split graphs:

In[1465]:=

```
IGSplitQ[g]
```

Out[1465]=

```
True
```

`IGThresholdQ` can be used directly on degree sequences:

In[1466]:=

```
IGThresholdQ@VertexDegree[g]
```

Out[1466]=

```
True
```

A threshold graph can be built by repeatedly adding either an isolated vertex, or a dominating vertex, i.e. a vertex that connects to all previous ones. The following function takes a specification in which `.` and `-` represent isolated and dominating vertices, respectively, and builds the corresponding graph.

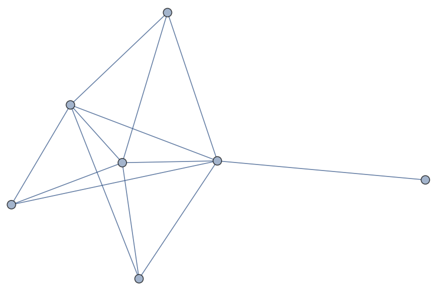
In[1467]:=

```
thresholdGraph[spec_String] :=
 With[{steps = Characters[spec]},
  Graph[Range@Length[steps],
   Join @@ Table[
     If[steps[[i]] === "-", UndirectedEdge[i, #] & /@ Range[i - 1], {}],
     {i, Length[steps]}
    ]
  ]
 ]
```

In[1468]:=

```
g = thresholdGraph["...--.-"]
```

Out[1468]=



In[1469]:=

```
IGThresholdQ[g]
```

Out[1469]=

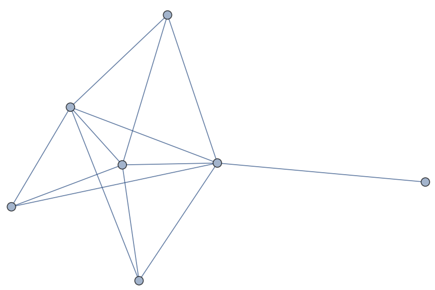
```
True
```

Degree sequences of threshold graphs have precisely one realization. Therefore, the same graph can be reconstructed from its degree sequence:

In[1470]:=

```
IGRealizeDegreeSequence@VertexDegree[g]
```

Out[1470]=



# Property handling and transformations

IGraph/M includes a set of functions that make it easy to extract vertex and edge properties (attributes), transform them with an arbitrary function, set their values based on the output of various functions such as `IGBetweenness`, or to copy values from one graph property into another.

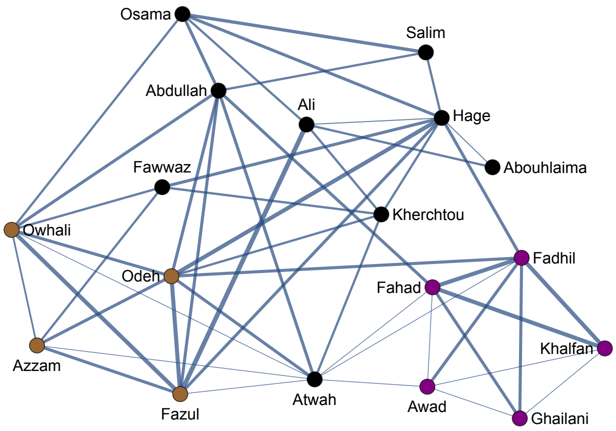
To simplify these tasks, IGraph/M's property handling framework takes a somewhat more restrictive view of graph properties than *Mathematica*'s built-ins. Edge and vertex properties are strictly distinguished, and it is assumed that when a property exists for one vertex (or edge), it also exists for all others. When this is not the case, the value `Missing["Nonexistent"]` is used.

Let us use the following example network to demonstrate the basic usage of these functions.

In[1471]:=

```
g = Graph[ExampleData[{"NetworkGraph", "EastAfricaEmbassyAttacks"}], ImageSize → Medium]
```

Out[1471]=




In[1472]:=

```
? IGVertexPropertyList
```

IGVertexPropertyList[graph] gives the list of available vertex properties in graph.

In[1473]:=

```
? IGEdgePropertyList
```

IGEdgePropertyList[graph] gives the list of available edge properties in graph.

Find what custom edge and vertex properties this graph has.

In[1474]:=

```
IGVertexPropertyList[g]
```

Out[1474]=

```
{FullName, Group, VertexCoordinates, VertexLabels,
 VertexShape, VertexShapeFunction, VertexSize, VertexStyle}
```

In[1475]:=

```
IGEdgePropertyList[g]
```

Out[1475]=

```
{EdgeShapeFunction, EdgeStyle, EdgeWeight}
```

In[1476]:=

```
? IGVertexProp
```

IGVertexProp[prop] is an operator that extracts the values of vertex property prop from a graph.

In[1477]:=

```
? IGEdgeProp
```

IGEdgeProp[prop] is an operator that extracts the values of edge property prop from a graph.

In[1478]:=

```
? IGEdgeVertexProp
```

IGEdgeVertexProp[prop] is an operator that extracts the vertex property prop for the vertex pair corresponding to each edge.

Extract the "Group" property of each node:

In[1479]:=

```
IGVertexProp["Group"][g]
```

Out[1479]=

```
{Planners, Planners, Planners, Planners, Planners, Planners, Planners, Planners,
 Nairobi Cell, Nairobi Cell, Nairobi Cell, Nairobi Cell, Planners, Dar es Salaam Cell,
 Dar es Salaam Cell, Dar es Salaam Cell, Dar es Salaam Cell, Dar es Salaam Cell}
```

Extract the weight of each edge:

In[1480]:=

```
IGEdgeProp[EdgeWeight][g]
```

Out[1480]=

```
{0.52, 0.36, 0.48, 0.48, 0.36, 0.36, 0.36, 0.36, 0.36, 0.16, 0.72, 0.16, 0.36, 0.36, 0.36, 0.36, 0.48,
 0.36, 0.36, 0.48, 0.48, 0.48, 0.48, 0.48, 0.64, 0.48, 0.48, 0.48, 0.64, 0.48, 0.48, 0.48, 0.64, 0.28,
 0.12, 0.48, 0.12, 0.12, 0.12, 0.12, 0.12, 0.64, 0.64, 0.48, 0.12, 0.64, 0.48, 0.48, 0.12, 0.12, 0.12}
```

In[1481]:=

```
? IGVertexMap
```

IGVertexMap[f, prop, graph] maps the function f to the vertex property list of property prop in graph.
IGVertexMap[f, prop –> pf, graph] maps the function f to
the values returned by pf[graph] and assigns the result to the vertex property prop.
IGVertexMap[f, prop –> {pf1, pf2, …}, graph] threads f over
{pf1[graph], pf2[graph], …} and assigns the result to the vertex property prop.
IGVertexMap[f, spec] represents an operator form of IGVertexMap that can be applied to a graph.

In[1482]:=

```
? IGEdgeMap
```

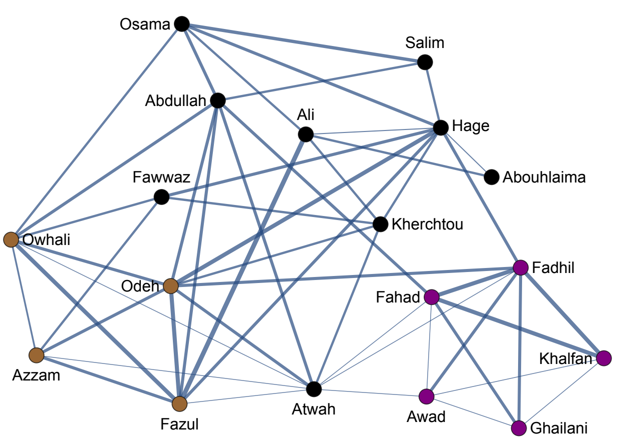
IGEdgeMap[f, prop, graph] maps the function f to the edge property list of property prop in graph.
IGEdgeMap[f, prop –> pf, graph] maps the function f to
the values returned by pf[graph] and assigns the result to the edge property prop.
IGEdgeMap[f, prop –> {pf1, pf2, …}, graph] threads f over
{pf1[graph], pf2[graph], …} and assigns the result to the edge property prop.
IGEdgeMap[f, spec] represents an operator form of IGEdgeMap that can be applied to a graph.

Show the value of the "FullName" custom property in tooltips.

In[1483]:=

```
IGVertexMap[# &, Tooltip → IGVertexProp["FullName"], g]
```

Out[1483]=



## Styling graphs according to stored or computed properties

In[1484]:=

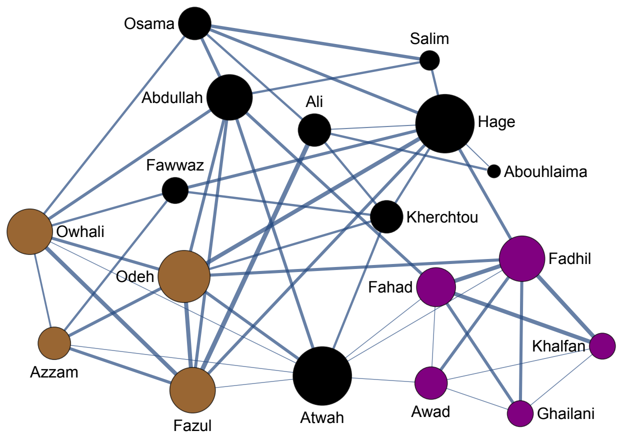
```
g = Graph[ExampleData[{"NetworkGraph", "EastAfricaEmbassyAttacks"}], ImageSize → Medium];
```

Scale vertices according to degree:

In[1485]:=

```
IGVertexMap[0.1 # &, VertexSize → VertexDegree, g]
```

Out[1485]=



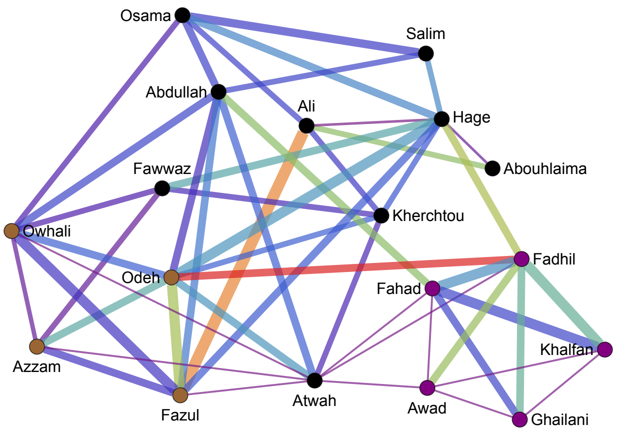

Let us colour edges by betweenness and set their thickness based on weight. Betweenness calculations treat high weight values as a “long distance”, thus we invert `EdgeWeight` before calculating the betweenness. To be able to use the original weights, we save them into a new `"weight"` property.

Calls to IGEdgeMap or IGVertexMap can be conveniently chained together using their operator form.

In[1486]:=

```
g //
 IGEdgeMap[ (* save original weight in "weight" property *)
   Identity, "weight" → IGEdgeProp[EdgeWeight]
  ] /*
  IGEdgeMap[ (* invert edge weights for betweenness calculation *)
   1 / # &, EdgeWeight
  ] /*
  IGEdgeMap[ (* thickness by original weight, colour by betweenness based on inverse weight *)
   Directive[AbsoluteThickness[9 #1], ColorData["Rainbow"][#2]] &,
   EdgeStyle → {IGEdgeProp["weight"], Rescale@* EdgeBetweennessCentrality}
  ]
```

Out[1486]=

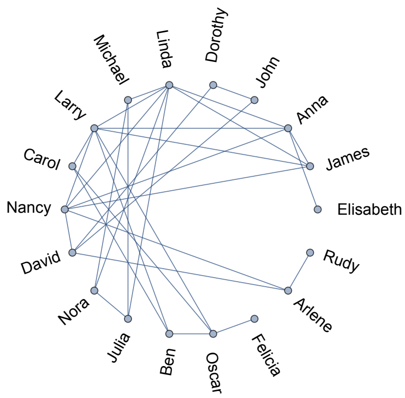


Label a graph with a circular layout:

In[1487]:=

```
IGVertexMap[
 Function[{name, coord},
  Placed[
   name,
   {{.5, .5}, -0.8 Normalize[coord] + {.5, .5}},
   Rotate[#, Mod[ArcTan @@ coord, Pi, -Pi / 2]] &
  ]
 ],
 VertexLabels → {VertexList, IGVertexProp[VertexCoordinates]},
 IGLayoutCircle[ExampleData[{"NetworkGraph", "FamilyGathering"}]]
]
```

Out[1487]=




Use edge weights as edge labels, and line up labels with edges:

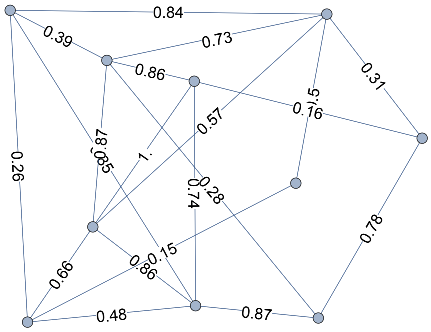
In[1488]:=

```
g = RandomGraph[{10, 20}, EdgeWeight → RandomReal[1, 20]];
```

In[1489]:=

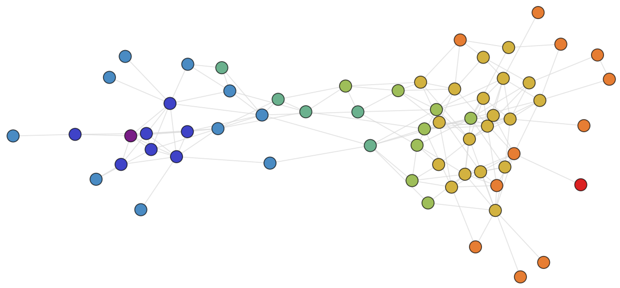
```
(* returns edge angle for each edge *)
edgeAngle[g_] :=
 With[{asc = AssociationThread[VertexList[g], GraphEmbedding[g]]},
  ArcTan @@ (asc[#1] - asc[#2]) & @@@ EdgeList[g]
 ]
```

In[1490]:=
```
IGLayoutDavidsonHarel[g] //
 IGEdgeMap[
  Function[{weight, angle},
   Placed[
    Style[NumberForm[weight, 2], Background → White],
    Center, Rotate[#, Mod[angle, Pi, -Pi / 2]] &
   ]
  ],
  EdgeLabels → {IGEdgeProp@EdgeWeight, edgeAngle}
 ]
```
Out[1490]=




Colour vertices based on their graph distance form a given vertex:

In[1491]:=
```
g = ExampleData[{"NetworkGraph", "DolphinSocialNetwork"}];
```
In[1492]:=
```
Graph[g, EdgeStyle → LightGray, VertexSize → 1, ImageSize → Medium] //
 IGVertexMap[
  ColorData["Rainbow"], VertexStyle → (Rescale@First@IGDistanceMatrix[#, {"Feather"}] &)
 ]
```
Out[1492]=



Colour the vertices of an annotated bipartite disease-gene graph based on whether they represent diseases or genes.

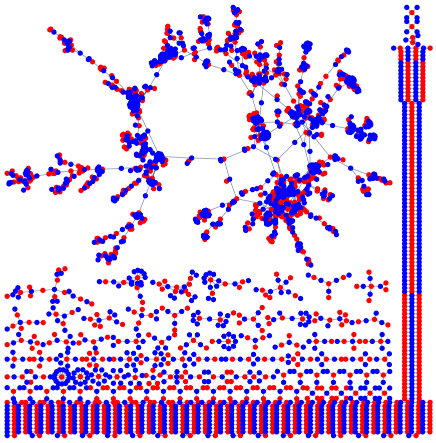
In[1493]:=
```
g = ExampleData[{"NetworkGraph", "BipartiteDiseasomeNetwork"}];
```
There are two types of vertices:

In[1494]:=
```
IGVertexProp["Type"][g] // Union
```
Out[1494]=
```
{Disease, Entrez}
```

In[1495]:=

```
IGVertexMap[
 <|"Disease" → Red, "Entrez" → Blue|>,
 VertexStyle → IGVertexProp["Type"],
 g
]
```

Out[1495]=

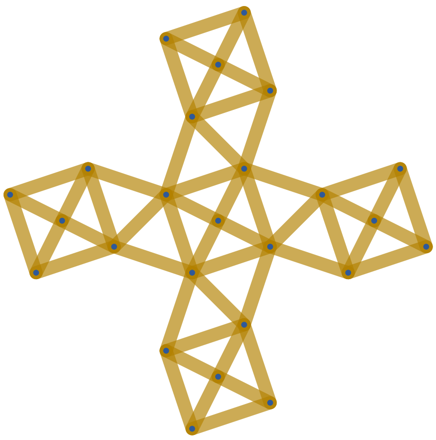


Compute the edge weights of a spatially embedded graph as lengths, then colour edges based on this value.

In[1496]:=

```
g = IGMeshCellAdjacencyGraph[
  IGLatticeMesh["Pinwheel", Disk[{0, 0}, 4]], 2,
  VertexCoordinates → Automatic,
  GraphStyle → "ThickEdge", EdgeStyle → Opacity[2 / 3]
 ]
```

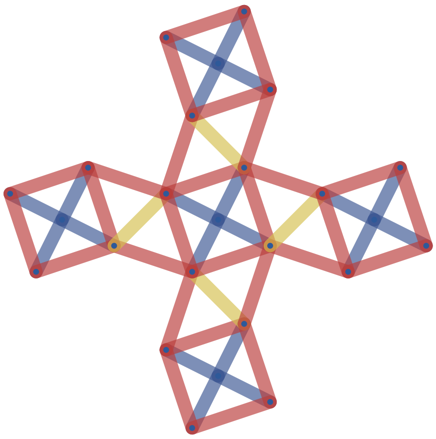
Out[1496]=



In[1497]:=

```
g //
 IGEdgeMap[Apply[EuclideanDistance], EdgeWeight → IGEdgeVertexProp[VertexCoordinates]] /*
  IGEdgeMap[ColorData["DarkRainbow"], EdgeStyle → Rescale@*IGEdgeProp[EdgeWeight]]
```

Out[1497]=

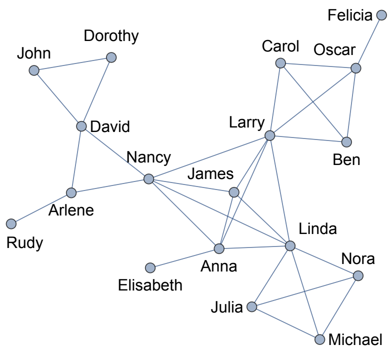


## Style social network by gender

In[1498]:=

```
g = ExampleData[{"NetworkGraph", "FamilyGathering"}]
```

Out[1498]=



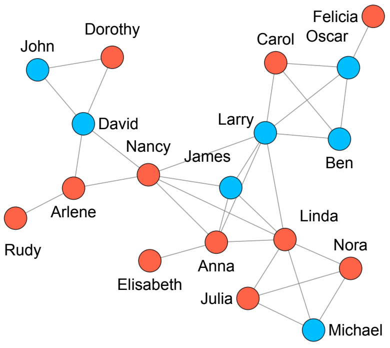

In[1499]:=

```
g = IGVertexMap[
   Interpreter["GivenName"][#]["Gender"] &,
   "gender" → VertexList,
   g
  ];
```

In[1500]:=

```
IGVertexMap[
 <| male GENDER → ■, female GENDER → ■|>,
 VertexStyle → IGVertexProp["gender"],
 Graph[g, EdgeStyle → Gray, VertexSize → Large]
]
```

Out[1500]=



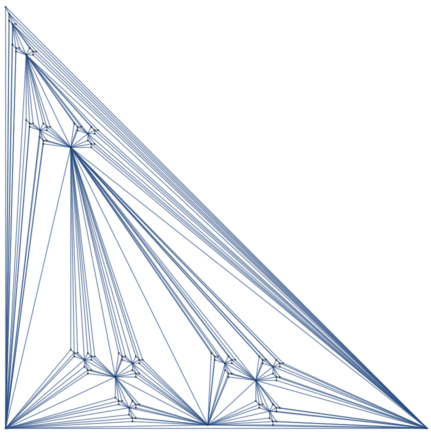

## Transform vertex coordinates

Transform the vertex coordinates of a graph to obtain a more pleasing layout:

In[1501]:=

```
g = Graph[GraphData[{"Apollonian", 5}, "EdgeList"], GraphLayout → "PlanarEmbedding"]
```

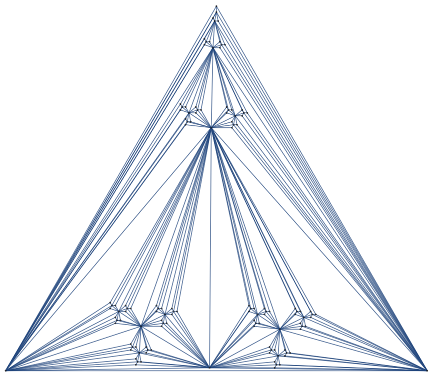
Out[1501]=



In[1502]:=

```
IGVertexMap[AffineTransform[{{1, 1/2}, {0, Sqrt[3]/2}}], VertexCoordinates, g]
```

Out[1502]=

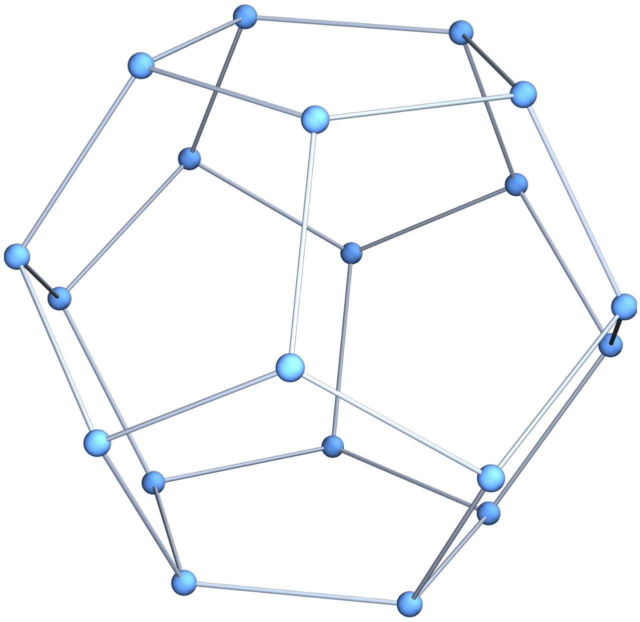


Project coordinates from the sphere to the plane using stereographic projection:

In[1503]:=

```
g = Graph3D[GraphData["DodecahedralGraph", "EdgeList"]]
```

Out[1503]=



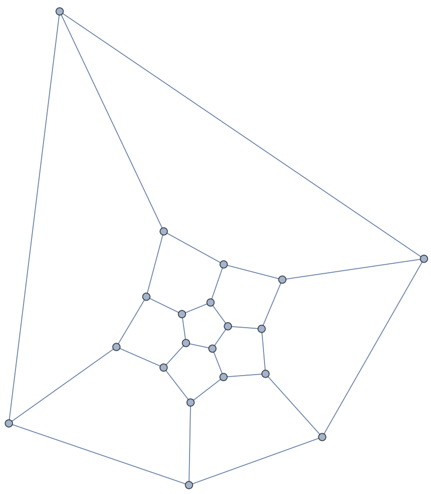
In[1504]:=

```
project =
  CoordinateTransform[
    {"Standard" → "Stereographic", {"Sphere", 1}},
    Rest@CoordinateTransform["Cartesian" → "Spherical", #]
   ] &;
```

In[1505]:=

```
Graph[
 IGVertexMap[project, VertexCoordinates → Standardize@*IGVertexProp[VertexCoordinates], g],
 GraphLayout → {"Dimension" → 2}
]
```

Out[1505]=



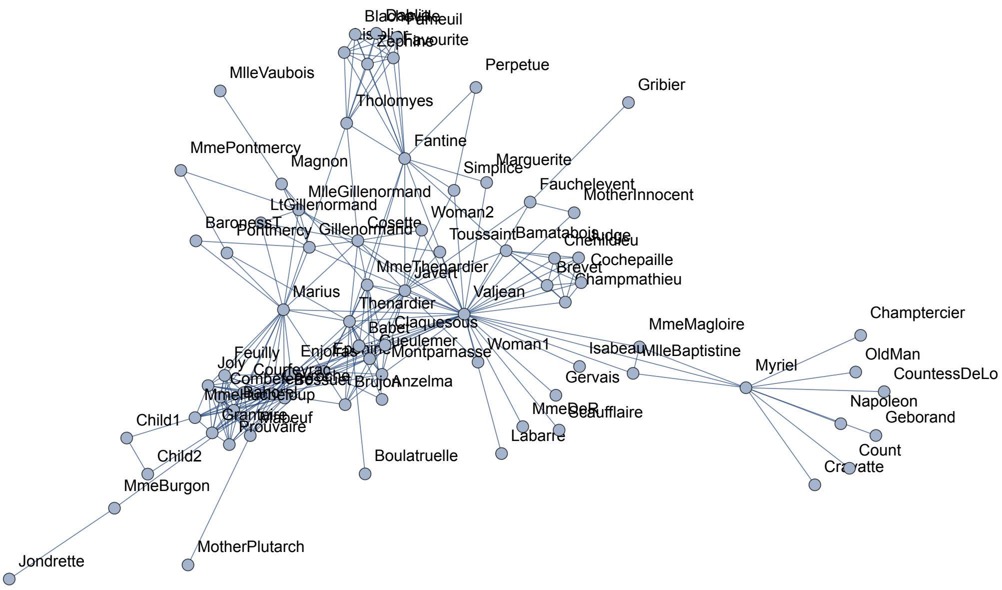
## Copy one property into another

Let us import this network:

In[1506]:=

```
g = Import["http://networkdata.ics.uci.edu/data/lesmis/lesmis.gml", ImageSize → Large]
```

Out[1506]=



According to the description, this should be a weighted graph:

In[1507]:=

```
Import["http://networkdata.ics.uci.edu/data/lesmis/lesmis.txt"]
```

Out[1507]=

```
The file lesmis.gml contains the weighted network of coappearances of
characters in Victor Hugo's novel "Les Miserables".  Nodes represent
characters as indicated by the labels and edges connect any pair of
characters that appear in the same chapter of the book.  The values on the
edges are the number of such coappearances.  The data on coappearances were
taken from D. E. Knuth, The Stanford GraphBase: A Platform for
Combinatorial Computing, Addison-Wesley, Reading, MA (1993).
```

But as imported by *Mathematica*, it is not edge-weighted:

In[1508]:=

```
IGEdgeWeightedQ[g]
```

Out[1508]=

```
False
```

This is because the edge weights are imported into the `"value"` property instead of the standard `EdgeWeight`:

In[1509]:=

```
IGEdgePropertyList[g]
```

Out[1509]=

```
{value, EdgeShapeFunction, EdgeStyle}
```

Copy the values of one property into another:

In[1510]:=

```
g = IGEdgeMap[# &, EdgeWeight → IGEdgeProp["value"], g];
```

Now we have a weighted graph:

In[1511]:=

```
IGEdgeWeightedQ[g]
```

Out[1511]=

```
True
```

# Matrix functions

## IGKirchhoffMatrix

In[1512]:=

```
? IGKirchhoffMatrix
```

IGKirchhoffMatrix[graph] gives the Kirchhoff matrix, also known as Laplacian matrix of graph.
IGKirchhoffMatrix[graph, "In"] will place the in–degrees on the diagonal instead of the out–degrees.

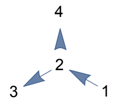
The Kirchhoff matrix of a graph is defined as

$$K_{i,j} = \begin{cases} -a_{i,j}, \text{ where } a_{i,j} \text{ is the number of } i \to j \text{ connections} & i \neq j \\ -\sum_{k \neq i} K_{k,i} & i = j \end{cases}$$

In other words, non-diagonal entries are the negative of the adjacency matrix and diagonal entries are equal to the out-degree. Rows sum up to zero.

The built-in `KirchoffMatrix` function uses the total degree on the diagonal even if the input graph is directed, making it unsuitable for many of the usual operations done with Kirchhoff matrices.

In[1513]:=

**`g =`**  **`;`**



In[1514]:=

```
KirchhoffMatrix[g] // MatrixForm
```

Out[1514]//MatrixForm=

$$\begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 3 & -1 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

By default, the diagonal contains the out-degrees, and rows sum to zero. This can also be requested explicitly using `IGKirchhoffMatrix[g, "Out"]`.

In[1515]:=

```
IGKirchhoffMatrix[g] // MatrixForm
```

Out[1515]//MatrixForm=

$$\begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

`IGKirchhoffMatrix[g, "In"]` will place the in-degrees on the diagonal, so that the columns will sum to zero.

In[1516]:=

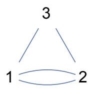
```
IGKirchhoffMatrix[g, "In"] // MatrixForm
```

Out[1516]//MatrixForm=

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

Unlike the built-in `KirchhoffMatrix`, `IGKirchoffMatrix` takes into account edge multiplicities.

In[1517]:=

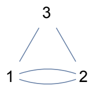
**`KirchhoffMatrix[`**  **`] // MatrixForm`**



Out[1517]//MatrixForm=

$$\begin{pmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{pmatrix}$$

In[1518]:=

**`IGKirchhoffMatrix[`**  **`] // MatrixForm`**



Out[1518]//MatrixForm=

$$\begin{pmatrix} 3 & -2 & -1 \\ -2 & 3 & -1 \\ -1 & -1 & 2 \end{pmatrix}$$

## IGJointDegreeMatrix

In[1519]:=

```
? IGJointDegreeMatrix
```

IGJointDegreeMatrix[graph] gives the joint degree matrix of graph. Element
i,j of the matrix contains the number of edges connecting degree−i and degree−j vertices.
IGJointDegreeMatrix[graph, d] gives the d by d joint degree matrix of graph, up to degree d.
IGJointDegreeMatrix[graph, {dOut, dIn}] gives the dOut by dIn joint degree matrix of graph.

Entry $J_{ij}$ of the joint degree matrix $J$ is the number of edges connecting a degree $i$ and a degree $j$ vertex. For a directed graph, $J_{ij}$ is the number of edges from a vertex with out-degree $i$ to a vertex with in-degree $j$.

For an empty (i.e. edgeless) graph, `{{}}` is returned.

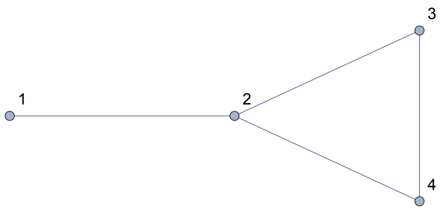
The available options are:

- `Normalized → True` will normalize the matrix by the sum of entries for directed graphs. For undirected graphs, the sum of upper triangular entries is used. Thus a matrix entry $J_{ij}$ can be interpreted as the probability that a randomly selected edge will connect vertices of degrees $i$ and $j$.

In[1520]:=

```
g = IGShorthand["1-2-3-4-2"]
```

Out[1520]=




In[1521]:=

```
jdm = IGJointDegreeMatrix[g];
MatrixForm[jdm, TableHeadings → Automatic]
```

Out[1522]//MatrixForm=

$$\left(\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 1 \\ 2 & 0 & 1 & 2 \\ 3 & 1 & 2 & 0 \end{array}\right)$$

The degree distribution (excluding zero-degree nodes) can be recovered as follows. The result array contains the number of nodes having each degree.

In[1523]:=

$$\frac{\texttt{Total[jdm] + Diagonal[jdm]}}{\texttt{Range@Max@VertexDegree[g]}}$$

Out[1523]=

```
{1, 2, 1}
```

Compute the degree distribution directly, for comparison.

In[1524]:=

```
Rest@BinCounts@VertexDegree[g]
```

Out[1524]=

```
{1, 2, 1}
```

Some other systems use a slightly different definition of the joint degree matrix for undirected graphs: the number of degree $i$ vertices connecting to degree $j$ vertices. Compared to the definition used here, this definition counts edges running between nodes of the same degree twice. To obtain this type of joint degree matrix, simply add the diagonal to the original matrix.

In[1525]:=

```
MatrixForm[jdm + DiagonalMatrix@Diagonal[jdm], TableHeadings → Automatic]
```

Out[1525]//MatrixForm=

$$\left(\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 1 \\ 2 & 0 & 2 & 2 \\ 3 & 1 & 2 & 0 \end{array}\right)$$

Multi-edges are supported.

In[1526]:=

```
MatrixForm[IGJointDegreeMatrix[
```

```
], TableHeadings → Automatic]
```

Out[1526]//MatrixForm=

$$\left(\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 2 \\ 3 & 1 & 2 & 0 \end{array}\right)$$

Self-loops are also supported. Note that `IGJointDegreeMatrix` counts loop edges twice when computing the vertex degree, just like `VertexDegree`. Thus the vertices of the below graph have degrees 1 and 3.

In[1527]:=

```
MatrixForm[IGJointDegreeMatrix[
```

```
], TableHeadings → Automatic]
```

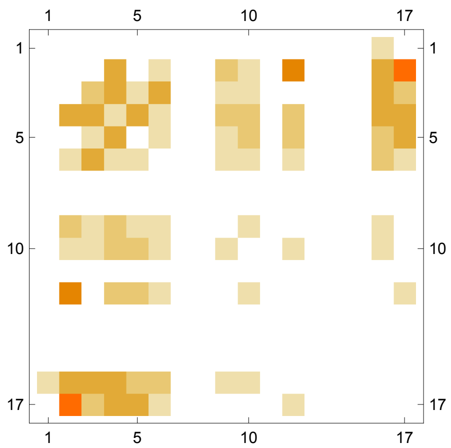
Out[1527]//MatrixForm=

$$\left(\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 \\ 3 & 1 & 0 & 1 \end{array}\right)$$

In[1528]:=

```
IGJointDegreeMatrix@ExampleData[{"NetworkGraph", "ZacharyKarateClub"}] // MatrixPlot
```

Out[1528]=

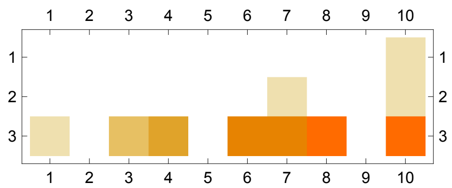



The joint degree matrix of a directed graph is not necessarily square.

In[1529]:=

```
IGBarabasiAlbertGame[15, 3] // IGJointDegreeMatrix // MatrixPlot
```

Out[1529]=

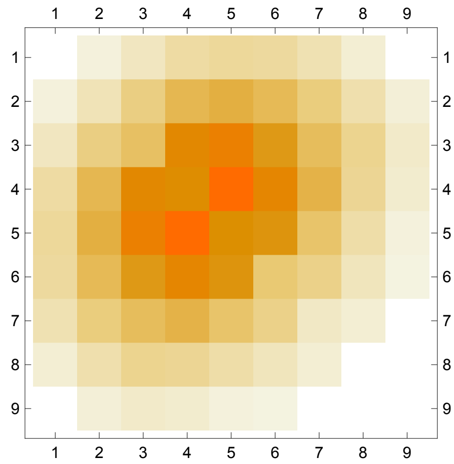



The second argument allows for obtaining joint degree matrices of a predictable size. This makes it convenient to operate together multiple joint degree matrices.

In[1530]:=

```
MatrixPlot@Mean@Table[
   IGJointDegreeMatrix[RandomGraph[{10, 20}], 9, Normalized → True],
   {1000}
  ]
```

Out[1530]=




## IGAdjacencyMatrixPlot

In[1531]:=

```
? IGAdjacencyMatrixPlot
```

IGAdjacencyMatrixPlot[graph] plots the adjacency matrix of graph.
IGAdjacencyMatrixPlot[graph, {v1, v2, …}] plots the adjacency matrix of the subgraph induced by the given vertices, using the specified vertex ordering.

`IGAdjacencyMatrixPlot` is based on `MatrixPlot`, but optimized for the convenient display of labelled adjacency matrices.

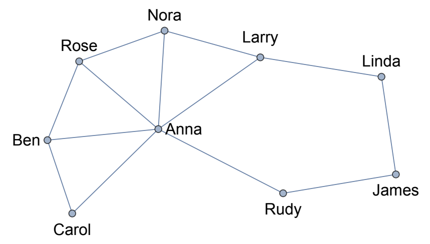
Available options:

- `EdgeWeight` sets the edge property to use for matrix elements. By default edge weights are used for weighted graphs. Set `EdgeWeight` → None to visualize the unweighted adjacency matrix even for a weighted graph.
- `"UnconnectedColor"` sets the colour to use to represent non-existent connections.
- `VertexLabels` controls how to label the matrix's columns and rows. Possible values:
  - `"Index"` uses row and column numbers. These are identical to vertex indices when the full adjacency matrix is plotted, but not when a partial matrix is plotted or if the vertices are re-ordered.
  - `"Name"` uses vertex names.
  - `Automatic` uses indices for large graphs and names for small ones. Use a list of rules to set different names for each vertex.
- `"RotateColumnLabels"` → `False` will not rotate columns labels.
- `Mesh` controls the drawing of grid lines. By default, a grid is drawn only for small graphs. Use `Mesh` → `All` to force drawing the grid.

`IGAdjacencyMatrixPlot` also accepts all standard `MatrixPlot` options.

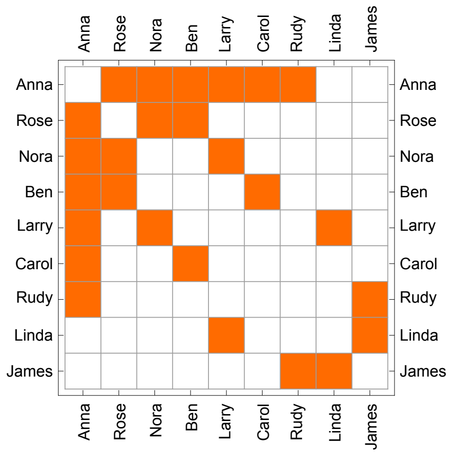
In[1532]:=
```
g = ExampleData[{"NetworkGraph", "Friendship"}]
```

Out[1532]=




In[1533]:=
```
IGAdjacencyMatrixPlot[g]
```

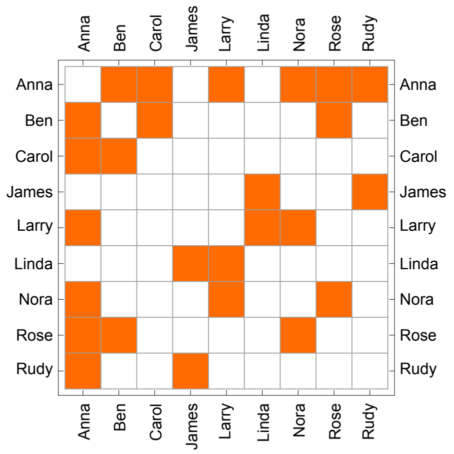
Out[1533]=




Reorder graph vertices before plotting.

In[1534]:=
```
IGAdjacencyMatrixPlot[g, Sort@VertexList[g]]
```

Out[1534]=

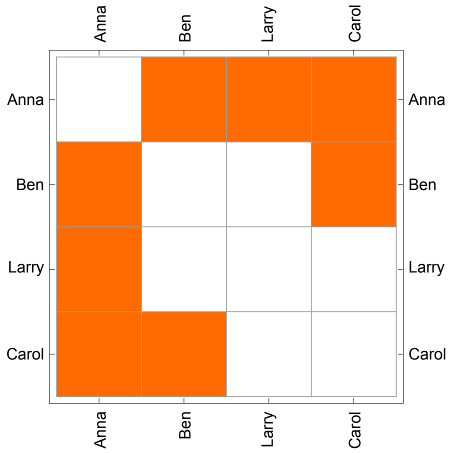


Plot a subgraph only.

In[1535]:=

```
IGAdjacencyMatrixPlot[g, {"Anna", "Ben", "Larry", "Carol"}]
```

Out[1535]=

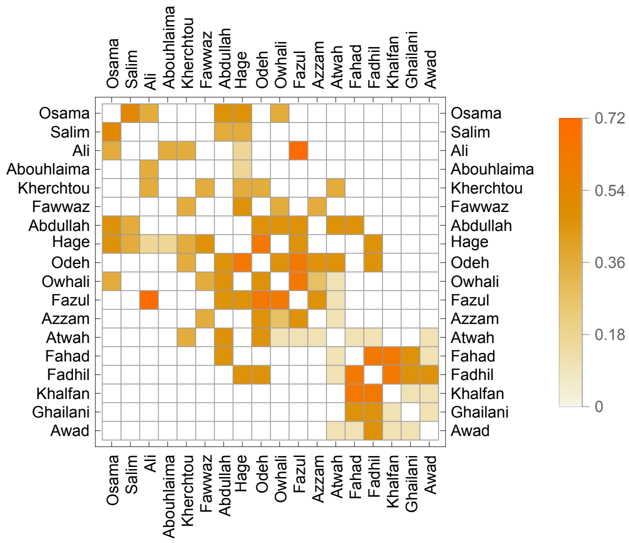



Plot a weighted adjacency matrix.

In[1536]:=

```
g = ExampleData[{"NetworkGraph", "EastAfricaEmbassyAttacks"}];
IGAdjacencyMatrixPlot[g, PlotLegends → Automatic, ImageSize → 300]
```

Out[1537]=




Use a different edge property than weights for the matrix entries.

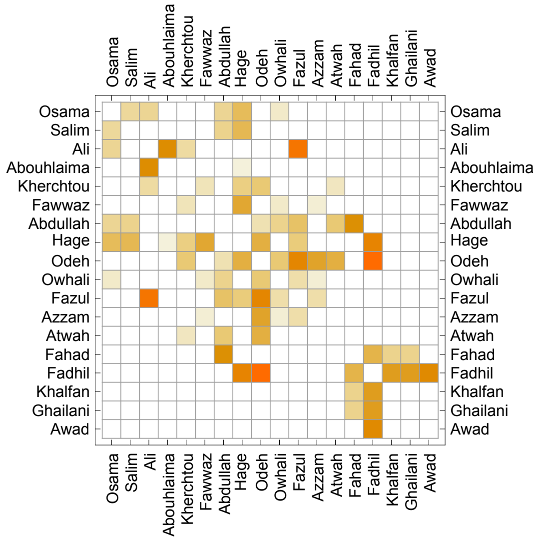
In[1538]:=

```
g2 = g // IGEdgeMap[1 / # &, EdgeWeight] //
    IGEdgeMap[# &, "Betweenness" → IGEdgeBetweenness];
```

In[1539]:=

```
IGAdjacencyMatrixPlot[g2, EdgeWeight → "Betweenness", ImageSize → 300]
```

Out[1539]=

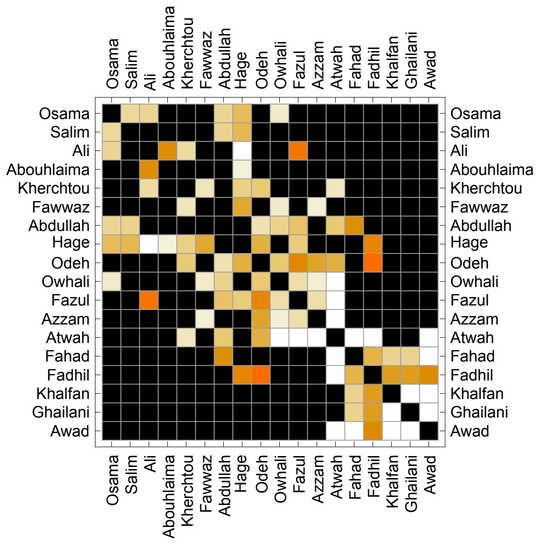



Control the style of matrix entries denoting the lack of a connection, to be able to distinguish them from zero entries.

In[1540]:=

```
IGAdjacencyMatrixPlot[g2, EdgeWeight → "Betweenness", "UnconnectedColor" → Black, ImageSize → 300]
```

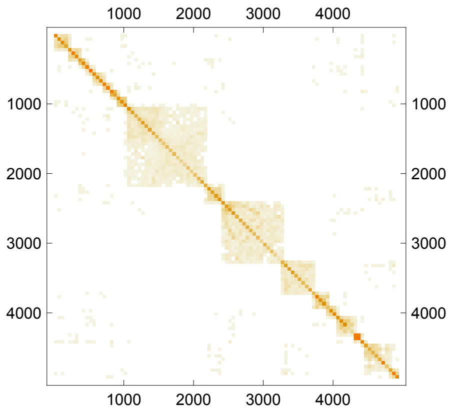
Out[1540]=



Plot the adjacency matrix of a very large network.

In[1541]:=
```
IGAdjacencyMatrixPlot[ExampleData[{"NetworkGraph", "PowerGrid"}]]
```

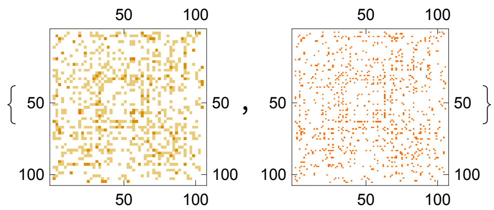
Out[1541]=




Large adjacency matrices, like the one above, are downsampled by default to improve readability. This can be controlled using the `MaxPlotPoints` option.

In[1542]:=
```
IGAdjacencyMatrixPlot[ExampleData[{"NetworkGraph", "USPoliticsBooks"}], MaxPlotPoints → #] & /@
 {Automatic, Infinity}
```

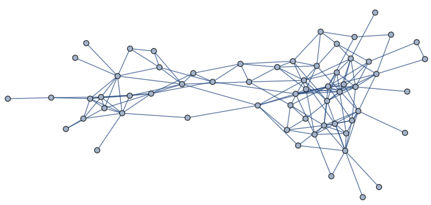
Out[1542]=




The vertex names and the grid are not shown by default for large graphs.

In[1543]:=
```
g = ExampleData[{"NetworkGraph", "DolphinSocialNetwork"}]
```

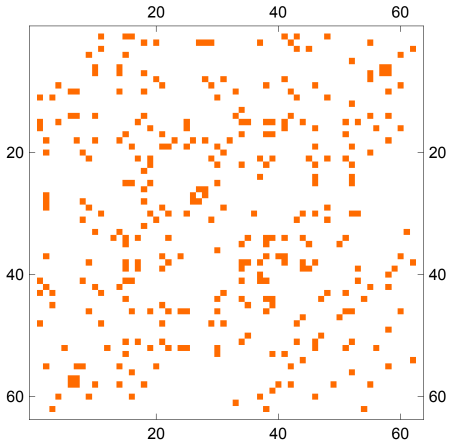
Out[1543]=



In[1544]:=

```
IGAdjacencyMatrixPlot[g]
```

Out[1544]=

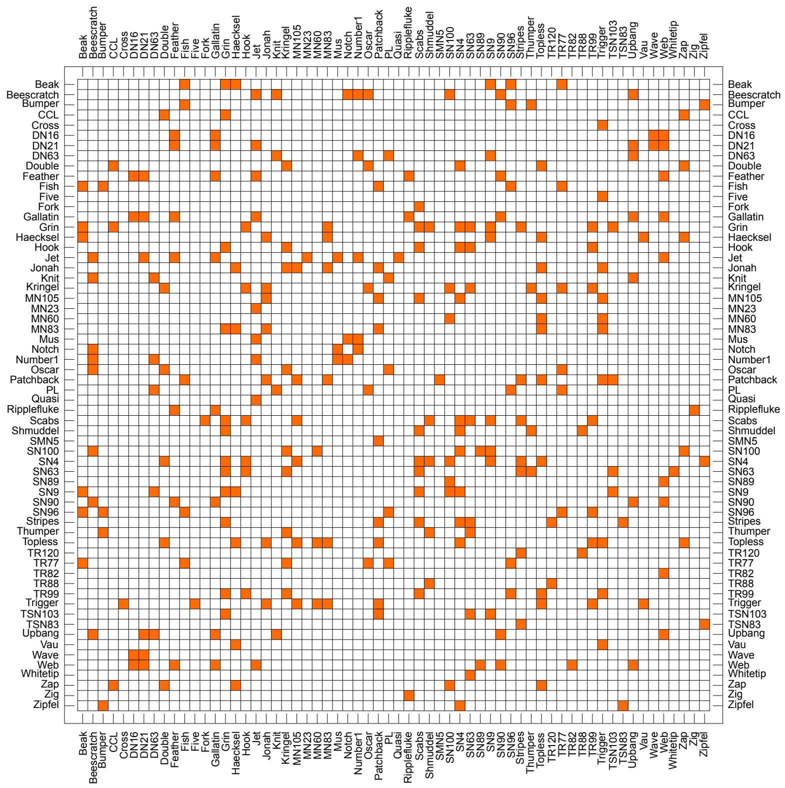


Force drawing vertex names and a grid regardless of the matrix size.

In[1545]:=

```
IGAdjacencyMatrixPlot[g, VertexLabels → "Name",
  Mesh → All, ImageSize → 440, FrameTicksStyle → Tiny, MeshStyle → Thin]
```

Out[1545]=

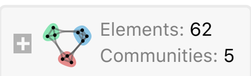


Reorder the adjacency matrix and draw grid lines to show community structure.

In[1546]:=

```
cl = IGCommunitiesEdgeBetweenness[g]
```

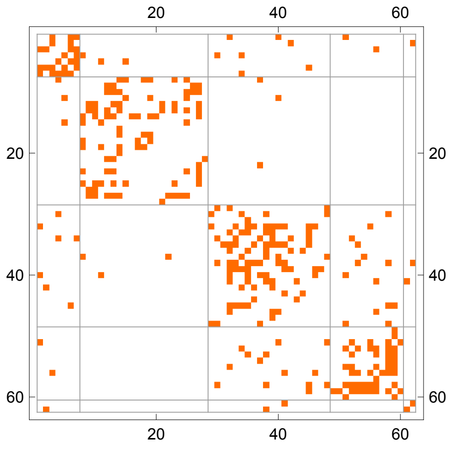
Out[1546]=

IGClusterData[  Elements: 62 Communities: 5 ]

In[1547]:=

```
IGAdjacencyMatrixPlot[g, Catenate@cl["Communities"],
 Mesh → ({#, #} &@FoldList[Plus, 0, Length /@ cl["Communities"]])]
```

Out[1547]=

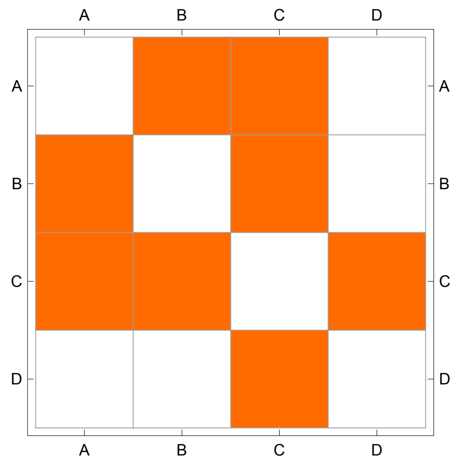



Avoid rotating vertex names when not necessary:

In[1548]:=

```
IGAdjacencyMatrixPlot[IGShorthand["A-B-C-D,A-C"], "RotateColumnLabels" → False]
```

Out[1548]=

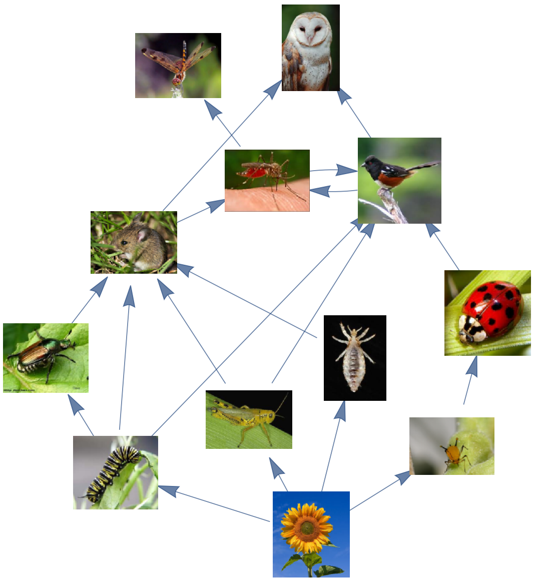


Use custom labels for vertices.

In[1549]:=

```
g = ExampleData[{"NetworkGraph", "SimpleFoodWeb"}]
```

Out[1549]=



In[1550]:=

```
names = Thread[VertexList[g] → (Show[#, ImageSize → 20] & /@ IGVertexProp[VertexShape][g])]
```

Out[1550]=

```
{Sunflower → , Aphid → , Ladybug → , Towhee → , Owl → , Grasshopper → ,
 Louse → , Caterpillar → , Beetle → , Mosquito → , Dragonfly → , Mouse → }
```

In[1551]:=

```
IGAdjacencyMatrixPlot[g, VertexLabels → names,
 "RotateColumnLabels" → False, ImageSize → Medium, ColorRules → {0 → White, 1 → Black}]
```

Out[1551]=

## IGZeroDiagonal

In[1552]:=

```
? IGZeroDiagonal
```

IGZeroDiagonal[matrix] replaces the diagonal of matrix with zeros.

`IGZeroDiagonal` replaces the diagonal of a matrix with zeros. It works on dense and sparse matrices, and supports non-square matrices. This function is particularly useful when constructing adjacency matrices that are to be converted to a graph.

In[1553]:=

```
mat = RandomReal[1, {6, 4}];
```

In[1554]:=

```
IGZeroDiagonal[mat] // MatrixForm
```

Out[1554]//MatrixForm=

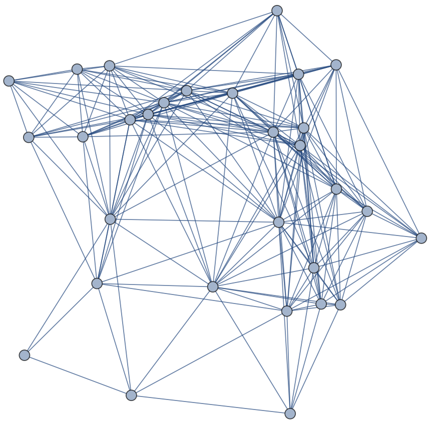
$$\begin{pmatrix} 0. & 0.600462 & 0.714719 & 0.219205 \\ 0.000765022 & 0. & 0.545862 & 0.152902 \\ 0.41223 & 0.216464 & 0. & 0.466637 \\ 0.7378 & 0.862239 & 0.296108 & 0. \\ 0.35644 & 0.629269 & 0.161126 & 0.129645 \\ 0.723072 & 0.270523 & 0.519175 & 0.562455 \end{pmatrix}$$

Connect those points in the plane whose Euclidean distance is less than 0.2, but do not connect each point with itself.

In[1555]:=

```
pts = RandomReal[1, {30, 2}];
AdjacencyGraph[
 IGZeroDiagonal@UnitStep[0.2 - DistanceMatrix[pts]],
 VertexCoordinates → pts
]
```

Out[1556]=

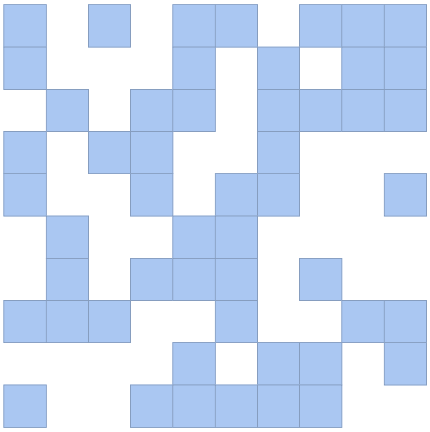


Connect each cell in a rectangular mesh to its Moore neighbours.

In[1557]:=

```
arr = RandomInteger[1, {10, 10}];
mesh = ArrayMesh[arr]
```

Out[1558]=



First, generate the square–vertex adjacency matrix.

In[1559]:=

```
mat = IGMeshCellAdjacencyMatrix[mesh, 2, 0]
```

Out[1559]=

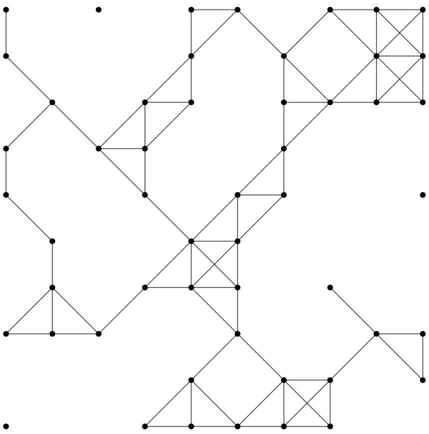
```
SparseArray[ Specified elements: 208
             Dimensions: {52, 112} ]
```

Then find the graph of squares adjacent through a corner point, but excluding self-adjacencies.

In[1560]:=

```
AdjacencyGraph[
 IGZeroDiagonal@Unitize[mat.Transpose[mat]],
 VertexCoordinates → PropertyValue[{mesh, 2}, MeshCellCentroid],
 GraphStyle → "BasicBlack"
]
```

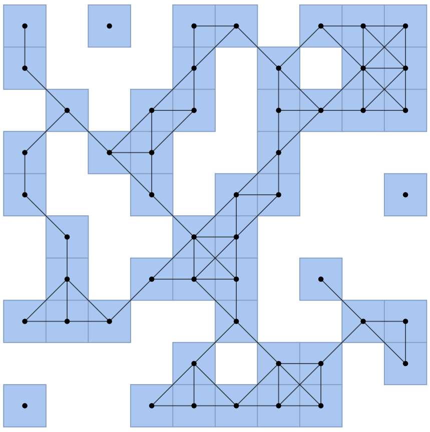
Out[1560]=



In[1561]:=

```
Show[mesh, %]
```

Out[1561]=



## IGTakeUpper and IGTakeLower

In[1562]:=

```
? IGTakeUpper
```

IGTakeUpper[matrix] extracts the elements of a matrix that are above the diagonal.

In[1563]:=

```
? IGTakeLower
```

IGTakeLower[matrix] extracts the elements of a matrix that are below the diagonal.

`IGTakeUpper` and `IGTakeLower` extract the above-diagonal and below-diagonal elements of a matrix. The matrix does not need to be square. The elements are always extracted row-by-row.

In[1564]:=

```
mat = Partition[Range[16], 4];
MatrixForm[mat]
```

Out[1565]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

In[1566]:=

```
IGTakeUpper[mat]
```

Out[1566]=

```
{2, 3, 4, 7, 8, 12}
```

In[1567]:=

```
IGTakeLower[mat]
```

Out[1567]=

```
{5, 9, 10, 13, 14, 15}
```

`IGTakeUpper` and `IGTakeLower` support sparse matrices. When given a sparse array as input, the result will also be a sparse array.

In[1568]:=
```
sa = SparseArray[RandomInteger[{1, 6}, {10, 2}] → RandomInteger[10, 10]];
MatrixForm[sa]
```
Out[1569]//MatrixForm=

$$\begin{pmatrix} 0 & 0 & 4 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 9 & 9 & 0 & 0 & 1 & 0 \end{pmatrix}$$

In[1570]:=
```
IGTakeUpper[sa]
```
Out[1570]=
```
SparseArray[ Specified elements: 3
             Dimensions: {15} ]
```

In[1571]:=
```
Normal[%]
```
Out[1571]=
```
{0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 1}
```

`IGTakeUpper` and `IGTakeLower` are optimized for performance.

In[1572]:=
```
mat = RandomReal[1, {1000, 1000}];
```
In[1573]:=
```
IGTakeUpper[mat]; // RepeatedTiming
```
Out[1573]=
```
{0.000947801, Null}
```

Compute the mean pairwise distance of a random point set.

In[1574]:=
```
pts = RandomReal[1, {100, 2}];
Mean@IGTakeUpper@DistanceMatrix[pts]
```
Out[1575]=
```
0.353733
```

# Import and export

IGraph/M provides importers and exporters for certain graph formats.

## Importing

In[1576]:=
```
? IGImport
```

IGImport[file] imports the graphs stored in file, inferring the format from the file extension.
IGImport[file, format] imports assuming the given file format. See $IGImportFormats for supported formats.

In[1577]:=
```
? IGImportString
```

IGImportString[string, format] imports the graphs stored in string assuming the given format.

Importing is done using `IGImport` and `IGImportString`, which work analogously to the built-in `Import` and `ImportString`. The supported export formats are listed in `$IGImportFormats`:

In[1578]:=
```
$IGImportFormats
```
Out[1578]=
```
{Graph6}
```

## Nauty / Graph6

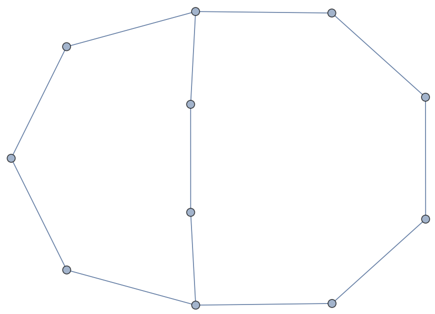
The Graph6, Sparse6 and Digraph6 formats are used by the `gtools` suite included with `nauty`. IGraph/M refers to this family of formats collectively as the “Nauty formats”. `gtools` includes many command line programs for generating, transforming and filtering graphs. For more information about `gtools`, see http://pallini.di.uniroma1.it/

The formal description of these formats is available on the home page of Brendan McKay at https://users.cecs.anu.edu.au/~bdm/data/formats.html.

As of version 12.1, *Mathematica* has built-in support for Graph6 and Sparse6, but not for Digraph6. IGraph/M provides support Digraph6, a unified interface to all three formats (with auto-detection of the specific sub-format), as well as significantly better performance.

To convert a single string to a graph, use `IGFromNauty`.

In[1579]:=
```
IGFromNauty["JR_IK?@?I?_"]
```
Out[1579]=

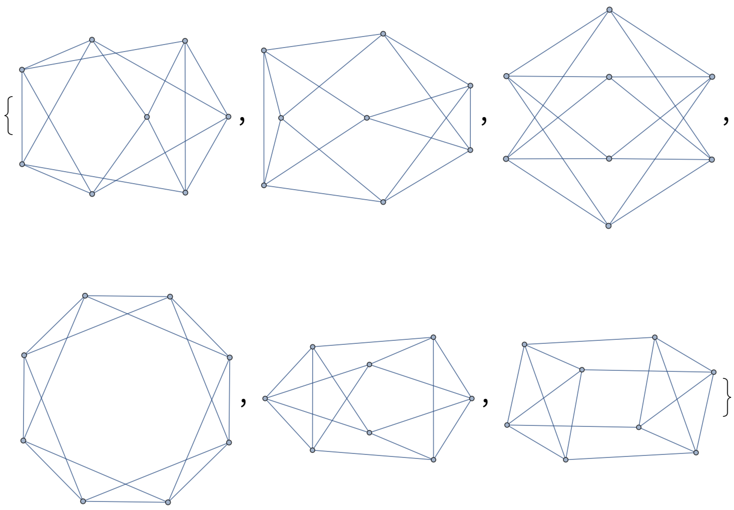


The purpose of `IGImport` and `IGImportString` is to read lists containing several graphs.

In[1580]:=

```
IGImportString["
G]kq]K
G]dq\\S
G]Ku]W
G[|akk
GS\\unO
G~`HW{", "Graph6"]
```

Out[1580]=



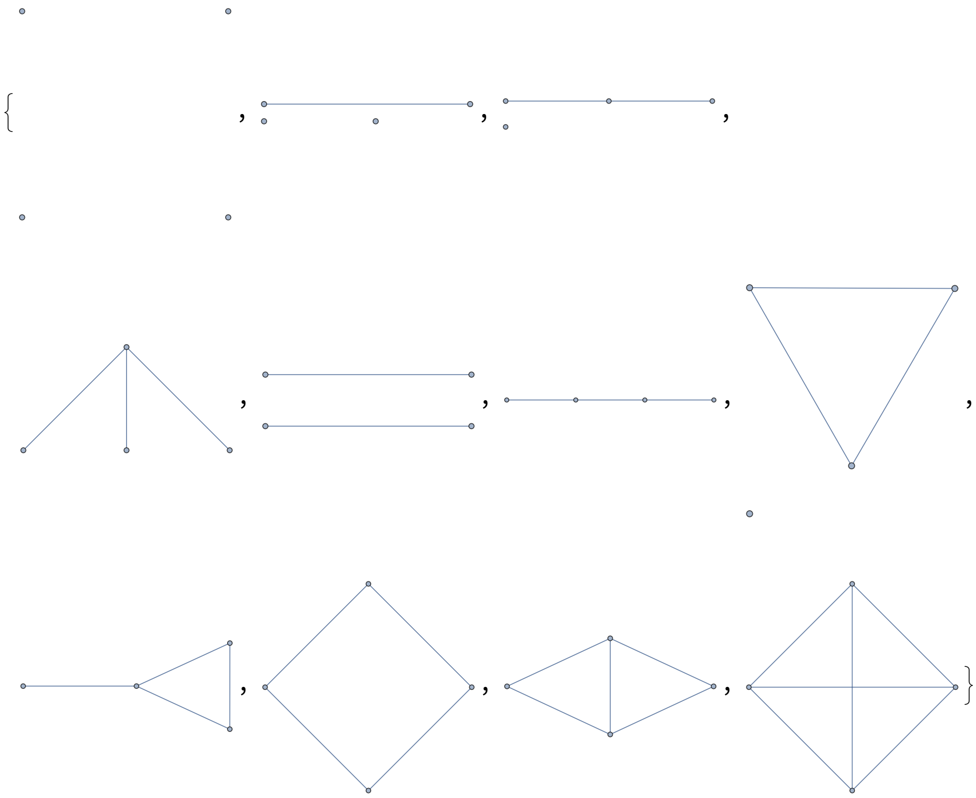
The following examples assume that the `gtools` programs are in a directory that is on the operating system's PATH environment variable. If necessary, specify the full path to each program.

Generate all non-isomorphic undirected graphs on 4 vertices.

In[1582]:=

```
IGImport["!geng 4", "Nauty"]
```

Out[1582]=

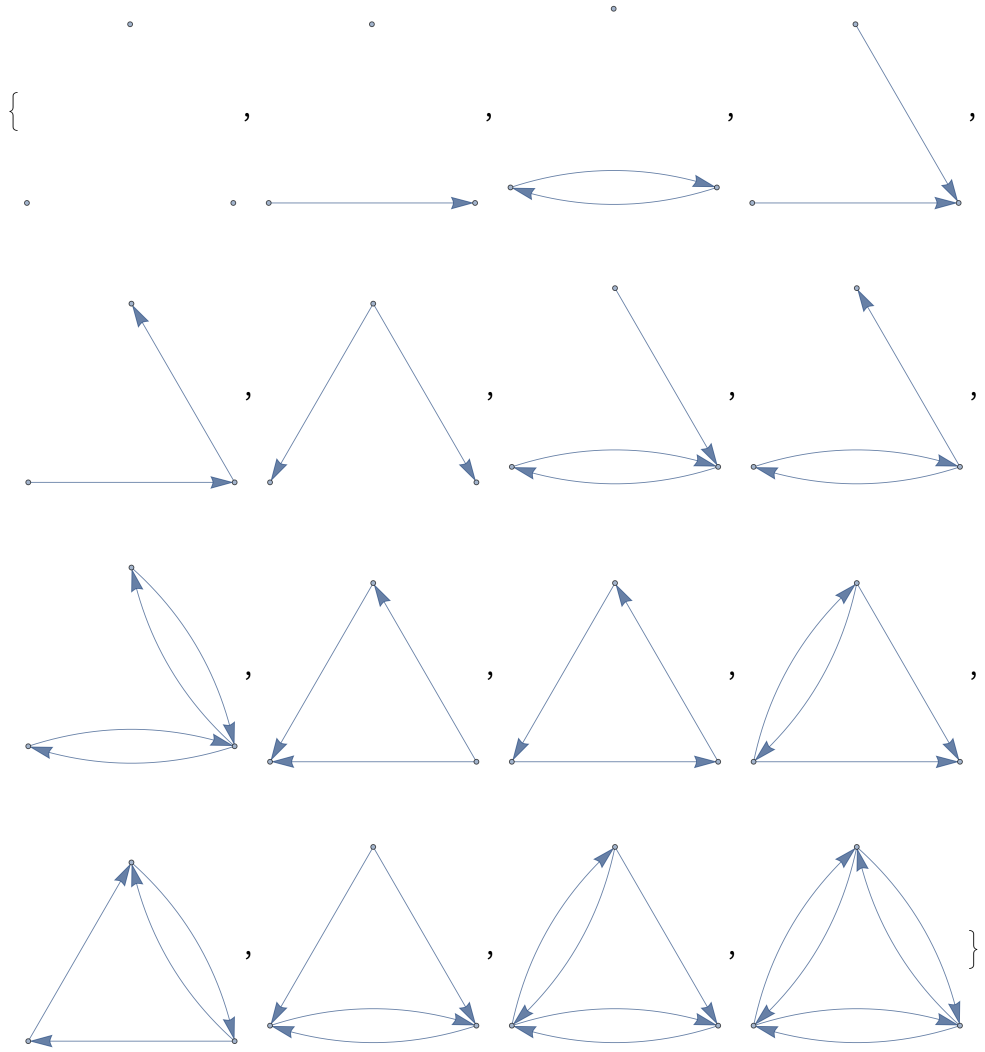


Generate all non-isomorphic directed graphs on 3 vertices.

In[1583]:=

```
IGImport["!geng 3 | directg", "Graph6", GraphLayout → "CircularEmbedding"]
```

Out[1583]=

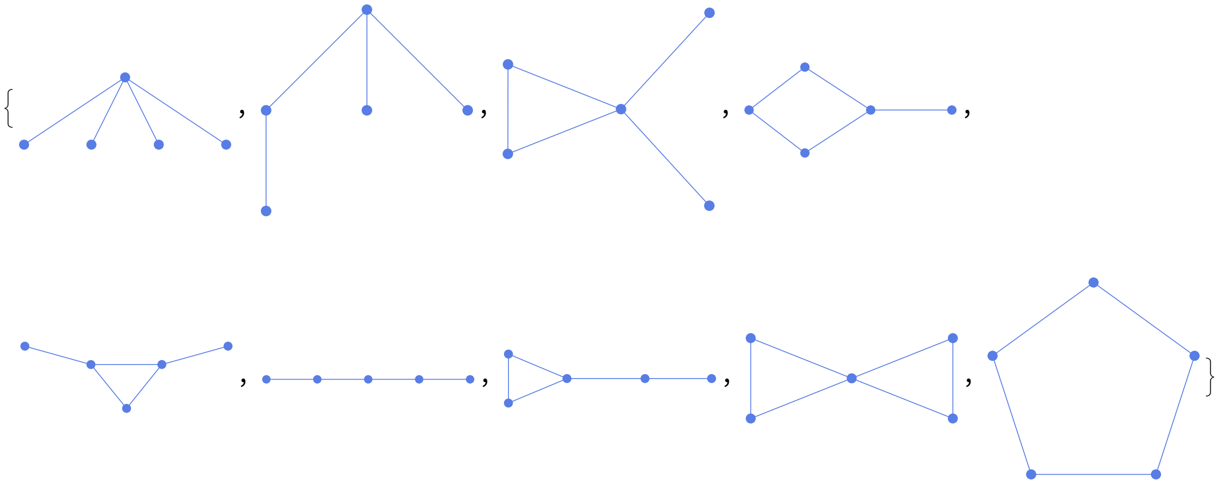


Find all non-isomorphic cactus graphs on 5 vertices. A cactus on $V$ vertices has between $V-1$ and $3(V-1)/2$ edges. Thus we instruct the geng program to only generate connected graphs with an edge count in this range.

In[1584]:=

```
Select[
 IGImport["!geng -c 5 4:6", "Graph6", GraphStyle → "Minimal"],
 IGCactusQ
]
```

Out[1584]=

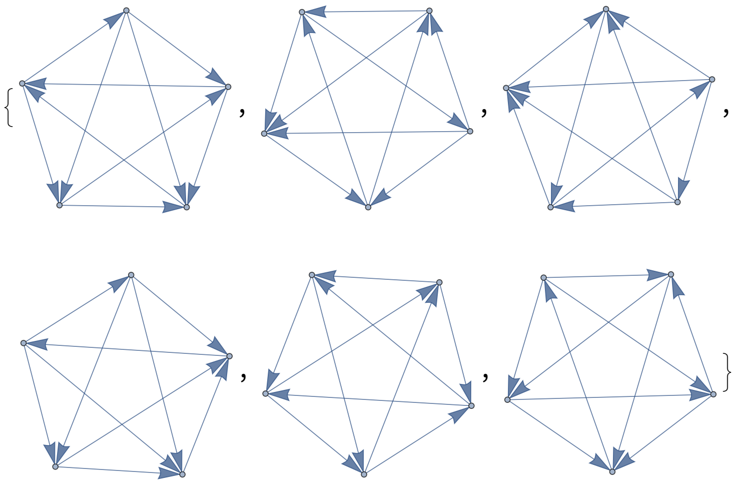


Generate all strongly connected tournaments on 5 vertices.

In[1585]:=

```
IGImport["!gentourng -c 5 -z", "Graph6"]
```

Out[1585]=

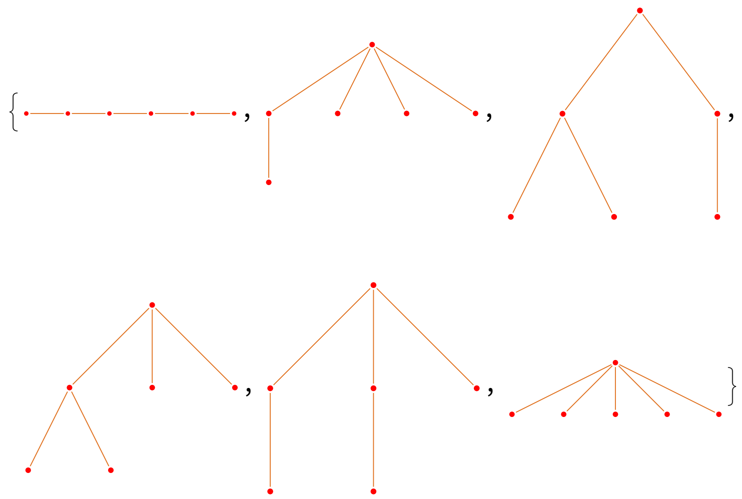


Generate all non-isomorphic trees on 6 vertices. `gengtreeg` outputs Sparse6 by default. `IGImport` detects this format automatically.

In[1586]:=

```
IGImport["!gentreeg 6", "Nauty", GraphStyle → "WarmColor"]
```

Out[1586]=



## Exporting

In[1587]:=

```
? IGExport
```

IGExport[file, graph] exports graph to file in a format inferred from the file extension.
IGExport[file, graph, format] exports graph to file in the given format. See $IGExportFormats for supported formats.

In[1588]:=

```
? IGExportString
```

IGExportString[graph, format] generates a string corresponding
to graph in the given format. See $IGExportFormats for supported formats.

Exporting is done using `IGExport` and `IGExportString`, which work analogously to the built-in `Export` and `ExportString`. The supported export formats are listed in `$IGExportFormats`:

In[1589]:=

```
$IGExportFormats
```

Out[1589]=

```
{GraphML}
```

## GraphML

As of *Mathematica* 13.0, the built-in `Export` function produces non-standard GraphML files that cannot be read by some other graph manipulation packages, such as the igraph library itself. `IGExport` provides a standards-compliant implementation.

In[1590]:=

```
IGExportString[
  ExampleData[{"NetworkGraph", "EurovisionVotes"}],
  "GraphML"
 ] // Short[#, 15] & (* avoid showing more than 15 lines *)
```

Out[1590]//Short=

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- created by IGraph/M, http://szhorvat.net/mathematica/IGraphM -->
<graphml xmlns='http://graphml.graphdrawing.org/xmlns'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://graphml.graphdrawing.org/xmlns
   http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd'>
 <key for='edge'
     id='e_EdgeWeight'
     attr.name='EdgeWeight'
     attr.type='long' />
 <graph id='Graph'
     edgedefault='directed'>
  <node id='Albania' />
  <node id='Andorra' />
  <node id='A … </edge>
  <edge source='United Kingdom'
      target='Malta'>
   <data key='e_EdgeWeight'>3</data>
  </edge>
  <edge source='United Kingdom'
      target='Netherlands'>
   <data key='e_EdgeWeight'>1</data>
  </edge>
  <edge source='United Kingdom'
      target='Norway'>
   <data key='e_EdgeWeight'>1</data>
  </edge>
  <edge source='United Kingdom'
      target='Sweden'>
   <data key='e_EdgeWeight'>2</data>
  </edge>
  <edge source='United Kingdom'
      target='Turkey'>
   <data key='e_EdgeWeight'>3</data>
  </edge>
 </graph>
</graphml>
```

# Utility functions

## Structural transformations

### IGUndirectedGraph

In[1591]:=

```
? IGUndirectedGraph
```

IGUndirectedGraph[graph, conv] converts a directed graph to undirected with the given conversion method: "Simple" creates a single edge between connected vertices; "All" creates an undirected edge for each directed one and may produce a multigraph; "Mutual" creates a single undirected edge only between mutually connected vertices.

`IGUndirectedGraph[g, method]` converts a directed graph to an undirected one, using the specified edge conversion method:

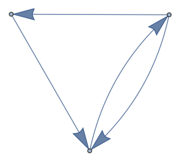
- `"Simple"` creates a single undirected edge between connected vertices
- `"All"` creates one undirected edge for each directed one. This may result in multiple edges between the same vertices.
- `"Mutual"` creates an edge only between mutually connected vertices.

If the graph was already undirected, it will not be changed.

`IGUndirectedGraph` is guaranteed to preserve both the vertex names and the vertex ordering of the original graph. The built-in `UndirectedGraph` has a bug where it sometimes relabels vertices.

**Warning:** As of IGraph/M 0.5, `IGUndirectedGraph` discards graph properties such as edge weights.

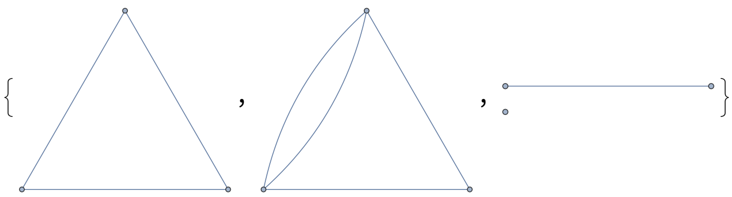
In[1592]:=

g =  ;

In[1593]:=

```
IGUndirectedGraph[g, #] & /@ {"Simple", "All", "Mutual"}
```

Out[1593]=

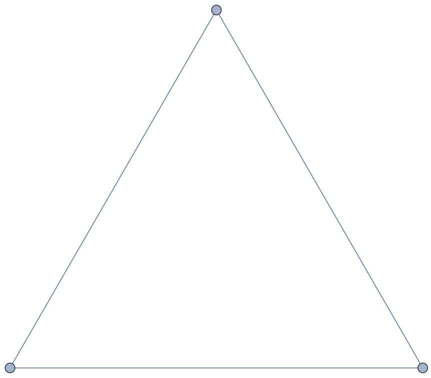


The default method is "Simple":

In[1594]:=

```
IGUndirectedGraph[g]
```

Out[1594]=

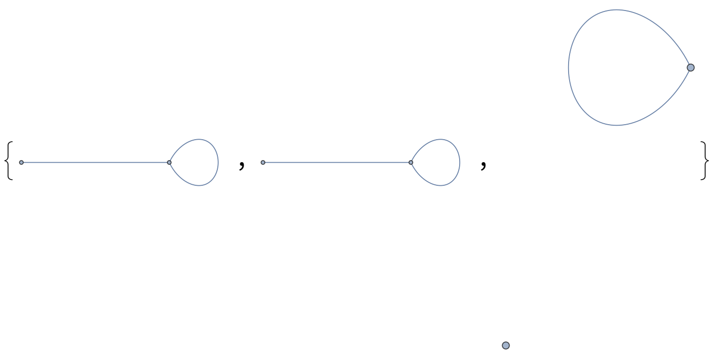


Self-loops are preserved by all methods:

In[1595]:=

```
IGUndirectedGraph[Graph[{1, 2}, {1 → 1, 1 → 2}], #] & /@ {"Simple", "All", "Mutual"}
```

Out[1595]=



## IGReverseGraph

In[1596]:=

```
? IGReverseGraph
```

IGReverseGraph[graph] reverses the directed edges in graph while preserving edge weights.

In *Mathematica* 11.3 and earlier, `ReverseGraph` does not correctly transfer graph properties such as edge weights to the result.

`IGReverseGraph` reverses the direction of each edge while preserving the following graph properties: `EdgeWeight`, `EdgeCapacity`, `EdgeCost`, `VertexWeight`, `VertexCapacity`. Other properties are discarded.

Undirected graphs are returned unmodified.

## IGSimpleGraph

In[1597]:=

```
? IGSimpleGraph
```

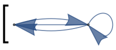
IGSimpleGraph[graph] converts graph to a simple
graph by removing self–loops and multi–edges, according to the given options.

`IGSimpleGraph` removes self-loops and collapses multi-edges into simple ones, as specified in the options.

The available options are:

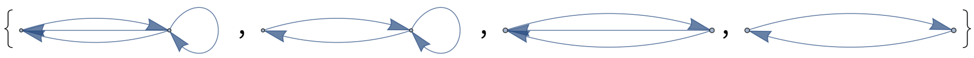
- `SelfLoops → True` keeps self-loops in the graph.
- `MultiEdges → True` keeps parallel edges in the graph.

In[1598]:=

**IGSimpleGraph[**  **, SelfLoops → #1, MultiEdges → #2] & @@@**

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

Out[1598]=



## IGDisjointUnion

In[1599]:=

```
? IGDisjointUnion
```

IGDisjointUnion[{g1, g2, …}] gives a disjoint union of the graphs. Each vertex of the result will be a pair consisting of the index of the graph originally containing it and the original name of the vertex.

`IGDisjointUnion` will combine the given graphs into a single graph having them as its connected components.

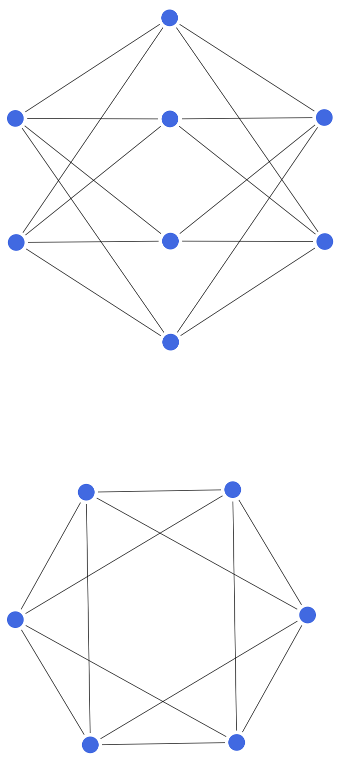
`IGDisjointUnion` differs from the built-in `GraphDisjointUnion` in several ways.

`IGDisjointUnion` takes the input graphs in a list instead of as separate arguments. It can also take graph options to apply to the final graph.

In[1600]:=

```
IGDisjointUnion[{TuranGraph[8, 2], TuranGraph[6, 3]},
 EdgeStyle → Black,
 VertexStyle → Directive[FaceForm[■], EdgeForm@Directive[Thick, □]], VertexSize → Medium]
```

Out[1600]=



IGDisjointUnion is considerably faster than GraphDisjointUnion when combining moderately sized networks.

In[1601]:=
```
graphs = RandomGraph[{300, 600}, 30];
```

In[1602]:=
```
IGDisjointUnion[graphs]; // RepeatedTiming
```

Out[1602]=
```
{0.0141539, Null}
```

In[1603]:=
```
GraphDisjointUnion @@ graphs; // RepeatedTiming
```

Out[1603]=
```
{0.537449, Null}
```

IGDisjointUnion does not support mixed graphs.

In[1604]:=
```
IGDisjointUnion[{Graph[{1 ⟷ 2}], Graph[{1 → 2}]}]
```

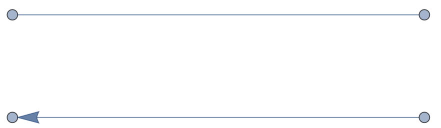
IGDisjointUnion: IGDisjointUnion does not support mixed graphs.

Out[1604]=
```
$Failed
```

Use GraphDisjointUnion instead.

In[1605]:=
```
GraphDisjointUnion[Graph[{1 ⟷ 2}], Graph[{1 → 2}]]
```

Out[1605]=



IGDisjointUnion will use vertex names of the form {gi, v}, where gi is an identifier of the original graph and v is the original name of the vertex. When the input is a list gi is the index of the original graph. When the input is an association, gi is its key.

In contrast, GraphDisjointUnion uses consecutive integers as vertex names. Use IndexGraph to obtain a similar result from the output of IGDisjointUnion.

In[1606]:=
```
g = IGDisjointUnion[<|"a" → CycleGraph[4], "b" → StarGraph[4]|>];
VertexList[g]
```

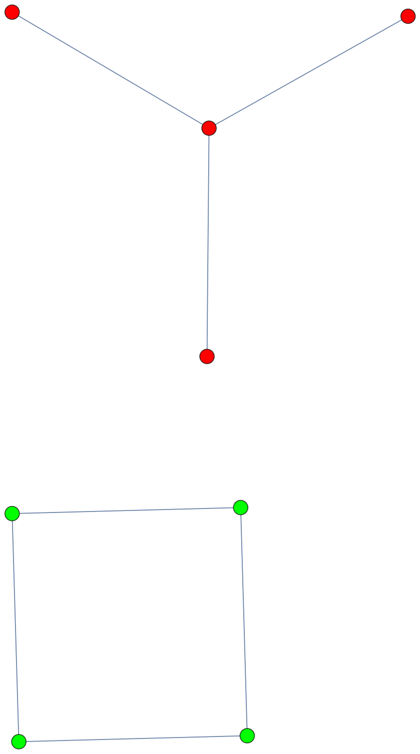
Out[1607]=
```
{{a, 1}, {a, 2}, {a, 3}, {a, 4}, {b, 1}, {b, 2}, {b, 3}, {b, 4}}
```

This allows for convenient further manipulation, or for copying over arbitrary properties stored in the original graphs.

In[1608]:=

```
Graph[g, VertexStyle → {{"a", _} → Green, {"b", _} → Red}]
```

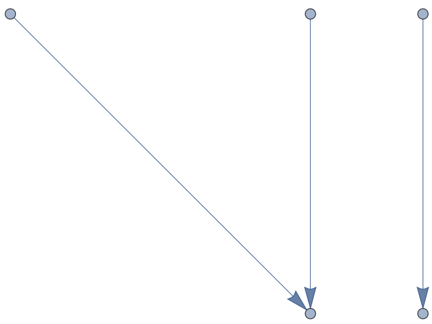
Out[1608]=



In[1609]:=

```
graphs = {
    Graph[{Property[1 → 2, "length" → 5]}],
    Graph[{Property[1 → 2, "length" → 3], Property[3 → 2, "length" → 2]}]
   };
IGDisjointUnion[graphs,
 Properties →
  {{gi_, v1_} ↔ {gi_, v2_} ⧴ {"length" → PropertyValue[{graphs[[gi]], v1 ↔ v2}, "length"]}}]
```

Out[1610]=



In[1611]:=

```
IGEdgeProp["length"][%]
```

Out[1611]=

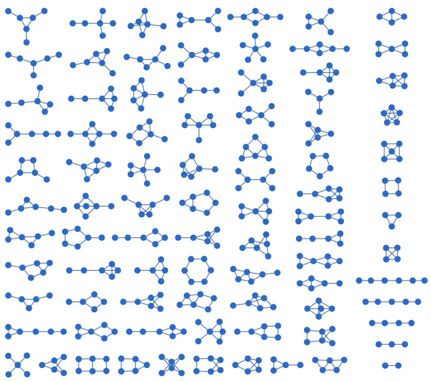
```
{5, 3, 2}
```

IGDisjointUnion is also practically useful for simply showing many small graphs together.

In[1612]:=

```
IGDisjointUnion[
 Select[IGGraphAtlas /@ Range[2, 150], IGConnectedQ],
 GraphStyle → "BasicBlue", VertexSize → 1
]
```

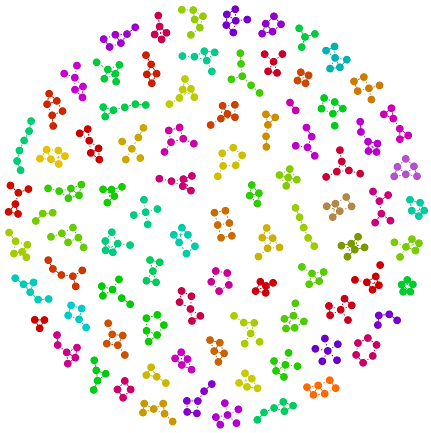
Out[1612]=



In[1613]:=

```
g = IndexGraph[%]; (* workaround for highlighting in Mathematica ≤ 11.2 *)
HighlightGraph[IGLayoutFruchtermanReingold[g], ConnectedGraphComponents[g]]
```

Out[1614]=



## IGOrientTree

In[1615]:=

```
? IGOrientTree
```

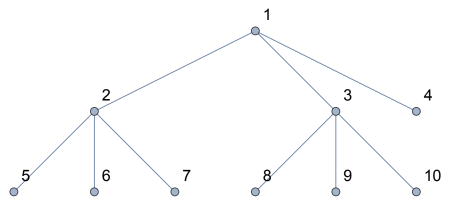
IGOrientTree[tree, root] orients the edges of an undirected tree so that they point away from the root. The vertex order is not preserved: vertices will be ordered topologically.
IGOrientTree[tree, root, "In"] orients the edges so that they point towards the root.

`IGOrientTree` creates an out-tree (also called an arborescence) out of an undirected tree. Graph properties are preserved, but the vertex ordering of the graph is changed.

In[1616]:=

```
g = KaryTree[10, 3, VertexLabels → "Name"]
```

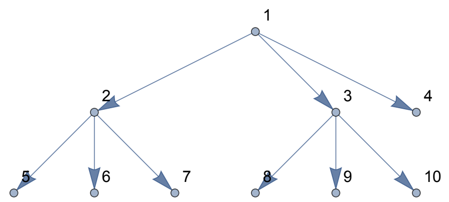
Out[1616]=




In[1617]:=

```
IGOrientTree[g, 1]
```

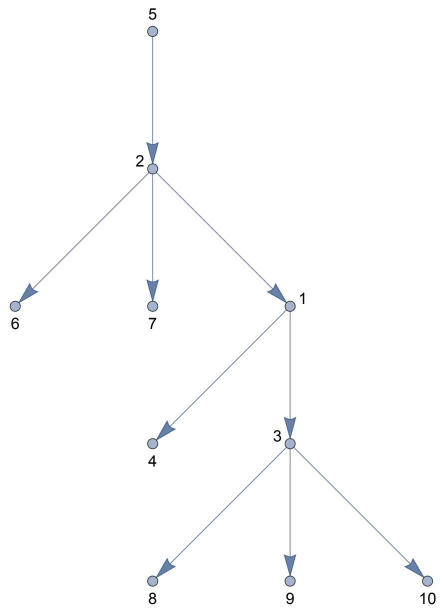
Out[1617]=




In[1618]:=

```
IGOrientTree[g, 5, GraphLayout → "LayeredDigraphEmbedding"]
```

Out[1618]=




The result is an out-tree, also called an arborescence.

In[1619]:=

```
IGTreeQ[%, "Out"]
```

Out[1619]=

```
True
```

Once the tree is made directed, it is easy to find its root and leaf nodes:

In[1620]:=

```
{IGSourceVertexList[%], IGSinkVertexList[%]}
```

Out[1620]=

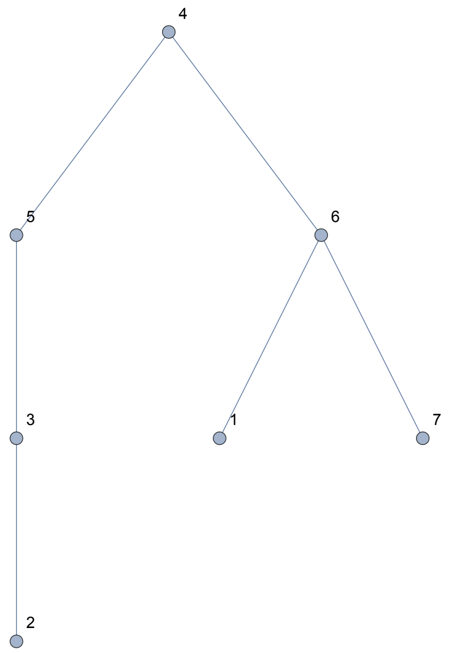
```
{IGSourceVertexList[True], IGSinkVertexList[True]}
```

Edges can be oriented towards or away from the root.

In[1621]:=

```
t = IGTreeGame[7, VertexLabels → Automatic]
```

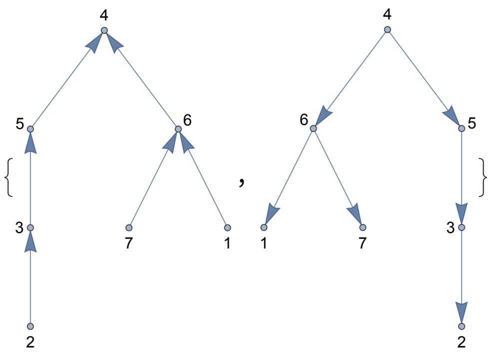
Out[1621]=




In[1622]:=

```
IGOrientTree[t, 4, #, GraphLayout → {"LayeredEmbedding", "RootVertex" → 4}] & /@ {"In", "Out"}
```

Out[1622]=




## IGTakeSubgraph

In[1623]:=

```
? IGTakeSubgraph
```

IGTakeSubgraph [graph, subgraph] keeps only those vertices and edges of graph which are also present in subgraph, while retaining all graph properties.
IGTakeSubgraph [graph, edges] uses an edge list as the subgraph specification.

`IGTakeSubgraph[graph, subgraph]` will effectively transfer graph properties of a larger graph onto its specified subgraph. It can be used in conjunction with other graph subsetting functions that do not retain some or all graph properties, such as `Subgraph`, `VertexDelete`, `NeighborhoodGraph`, etc.

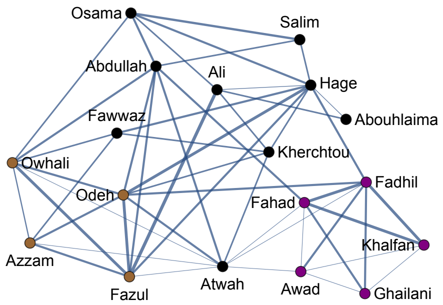
If only the edge weights need to be preserved, use `IGWeightedSubgraph` when possible. It offers better performance.

Take a subgraph of a larger graph while preserving all graph properties, including styling attributes.

In[1624]:=

```
g = ExampleData[{"NetworkGraph", "EastAfricaEmbassyAttacks"}]
```

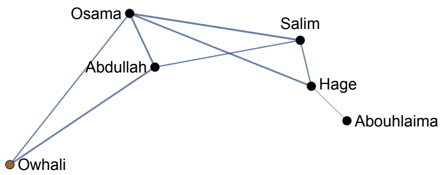
Out[1624]=




In[1625]:=

```
IGTakeSubgraph[g, Subgraph[g, {"Osama", "Salim", "Abdullah", "Hage", "Abouhlaima", "Owhali"}]]
```

Out[1625]=

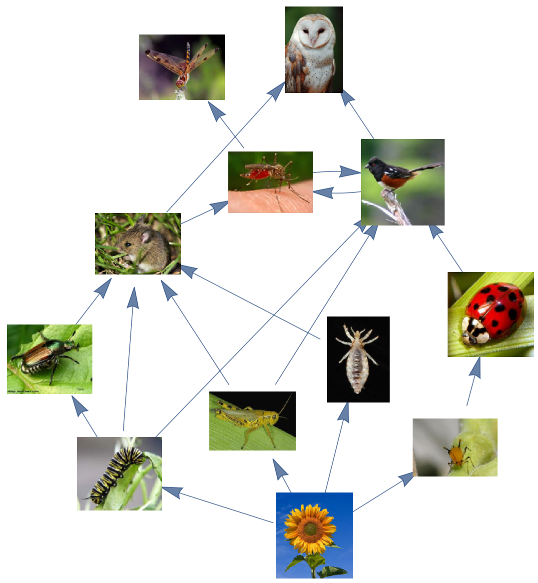



Show the neighbourhood graph of a vertex while preserving vertex shapes.

In[1626]:=

```
g = ExampleData[{"NetworkGraph", "SimpleFoodWeb"}]
```

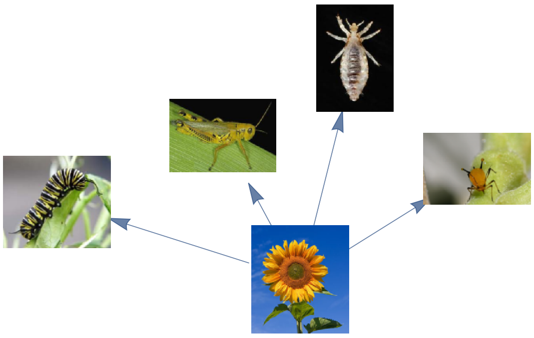
Out[1626]=



In[1627]:=

```
IGTakeSubgraph[g, NeighborhoodGraph[g, "Sunflower"]]
```

Out[1627]=

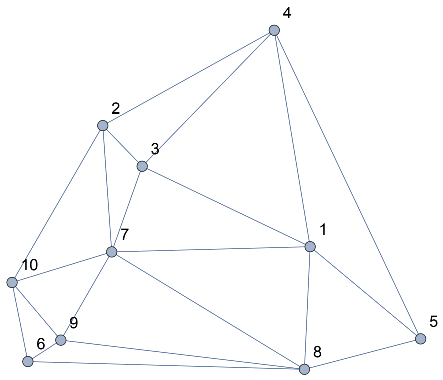


Take a subgraph of a mesh graph while preserving vertex coordinates.

In[1628]:=

```
g = IGMeshGraph[DelaunayMesh@RandomReal[1, {10, 2}], VertexLabels → "Name"]
```

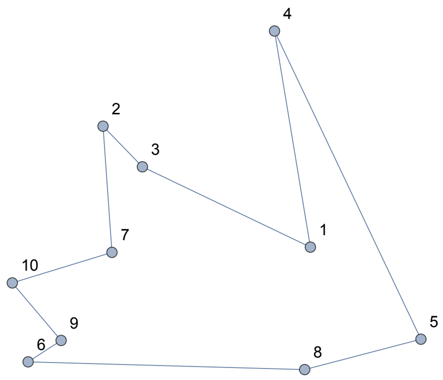
Out[1628]=




In[1629]:=

```
IGTakeSubgraph[g, First@FindHamiltonianCycle[g]]
```

Out[1629]=




## Graph editor

In[1630]:=

```
? IGGraphEditor
```

IGGraphEditor[] typesets to an interactive graph editor. Use CMD–click to add/remove vertices/edges.
IGGraphEditor[graph] uses the given graph as the starting point.

**Experimental:** This is experimental functionality that is likely to change significantly in the future.

`IGGraphEditor[]` typesets an interactive graph editor, which is convenient for creating small graphs interactively. To add or remove vertices, or remove edges, click while holding down ALT (Windows and Linux) or CMD (macOS). To add edges, click the first vertex to connect, then the second one.

Evaluating the editor using SHIFT-ENTER creates a standard `Graph` object.

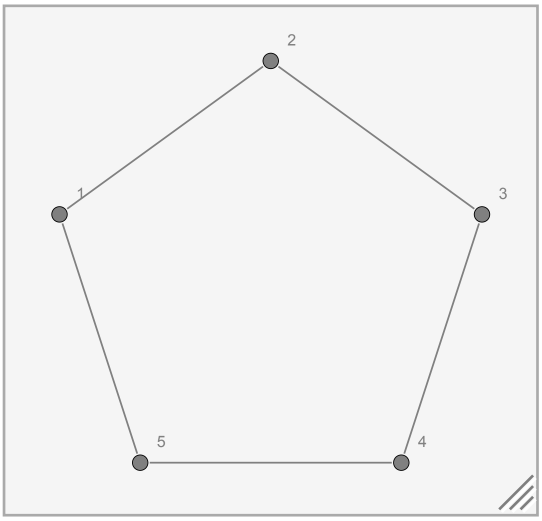
Available options:

- `DirectedEdges → True` creates a directed graph when no input graph is given.
- `"KeepVertexCoordinates" → False` will not preserve the vertex coordinates from the editor view.
- `"IndexGraph" → True` renumbers vertices using increasing integers, regardless of their original names in the input graph.
- `ImageSize` sets the editor size to the given width.
- `VertexLabels → "Name"` shows vertex labels in the editor.
- `VertexSize` sets the vertex size. Valid values are `Tiny`, `Small`, `Medium`, `Large` or a numeric value interpreted as a fraction of the editor view diagonal.
- `"PerformanceLimit"` sets the maximum number of graph elements (vertices and edges) that are allowed in the editor. The default is 450.
- `"SnapToGrid" → True` will snap vertices to points on a grid while dragging.
- `"CreateVertexSelects" → False` disables immediately selecting newly created vertices to add a connection.

The editor can be used to modify an existing graph:

In[1631]:=
```
IGGraphEditor[CycleGraph[5], VertexLabels → "Name"]
```

Out[1631]=




## Other utility functions

### IGIndexEdgeList

In[1632]:=
```
? IGIndexEdgeList
```

IGIndexEdgeList[graph] gives the edge list of graph in terms of vertex indices, as a packed array.

`IGIndexEdgeList` is useful for implementing graph processing functions in *Mathematica*, and is used internally by many IGraph/M functions that do not call the igraph library.

In[1633]:=
```
IGIndexEdgeList[Graph[{a, b, c}, {b ↔ c, c ↔ a}]]
```
Out[1633]=
```
{{2, 3}, {1, 3}}
```

In[1634]:=
```
Developer`PackedArrayQ[%]
```
Out[1634]=
```
True
```

`IGIndexEdgeList[g]` is faster than `EdgeList[g]` and usually much faster than `EdgeList@IndexGraph[g]`.

In[1635]:=
```
g = ExampleData[{"NetworkGraph", "CondensedMatterCollaborations"}];
```

In[1636]:=
```
{First@RepeatedTiming@EdgeList[g],
 First@RepeatedTiming@EdgeList@IndexGraph[g],
 First@RepeatedTiming@IGIndexEdgeList[g]}
```
Out[1636]=
```
{0.0110604, 0.301264, 0.0017866}
```

In[1637]:=
```
List @@@ Sort /@ EdgeList@IndexGraph[g] === Sort /@ IGIndexEdgeList[g]
```
Out[1637]=
```
True
```

A graph can be directly re-built from an index-based edge list.

In[1638]:=
```
Graph[{a, b, c}, {{2, 3}, {1, 3}}]
```
Out[1638]=



## IGSameGraphQ

In[1639]:=
```
? IGSameGraphQ
```

IGSameGraphQ[graph1, graph2] returns True if the given graphs have the same vertices and edges. Graph properties or edge and vertex orderings are not taken into account.

`IGSameGraphQ` checks if two graphs have the same vertex and edge set. Edge and vertex properties, as well as edge tags, are ignored.

In[1640]:=
```
IGSameGraphQ[IGShorthand["1-2-1", MultiEdges → True], Graph[{1 ↔ 2, 1 ↔ 2}]]
```
Out[1640]=
```
True
```

The vertex names must be the same in order for `IGSameGraphQ` to return `True`.

In[1641]:=
```
IGSameGraphQ[IGShorthand["A-B-A", MultiEdges → True], Graph[{1 ↔ 2, 1 ↔ 2}]]
```
Out[1641]=
```
False
```

The order of the edge and vertex lists does not matter.

In[1642]:=

```
IGSameGraphQ[Graph[{1 ↔ 2, 3 ↔ 4}], Graph[{4 ↔ 3, 1 ↔ 2}]]
```

Out[1642]=

```
True
```

For non-`Graph` expressions, `IGSameGraphQ` returns `False`.

In[1643]:=

```
IGSameGraphQ[1, 2]
```

Out[1643]=

```
False
```

## IGCanonicalLabeledGraph

In[1644]:=

```
? IGCanonicalLabeledGraph
```

IGCanonicalLabeledGraph[graph] canonicalizes the vertex and edge lists of a graph while preserving vertex names.

`IGCanonicalLabeledGraph` creates a canonical version of labelled graphs so that `IGCanonicalLabeledGraph[g1] === IGCanonicalLabeledGraph[g2]` holds precisely when `IGSameGraphQ[g1, g2]`.

This function discards all graph properties, as well as edge tags.

In[1645]:=

```
g1 = Graph[{4 ↔ 3, 1 ↔ 2}, VertexLabels → Automatic];
g2 = Graph[{1 ↔ 2, 3 ↔ 4}, VertexLabels → Automatic];

IGCanonicalLabeledGraph[g1] === IGCanonicalLabeledGraph[g1]
```

Out[1647]=

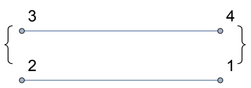
```
True
```

`IGCanonicalLabeledGraph` is useful in conjunction with DeleteDuplicatesBy.

In[1648]:=

```
DeleteDuplicatesBy[{g1, g2, g1}, IGCanonicalLabeledGraph]
```

Out[1648]=




## IGCanonicalEdgeList

In[1649]:=

```
? IGCanonicalEdgeList
```

IGCanonicalEdgeList[edges] canonicalizes an edge list.

`IGCanonicalEdgeList` canonicalizes an edge list in a way similar to `IGCanonicalLabeledGraph`. `IGSameGraphQ[g1, g2]` is equivalent to `IGCanonicalEdgeList@EdgeList[g1] === IGCanonicalEdgeList@EdgeList[g2]` when g1 and g2 have no isolated vertices.

This function discards edge tags.

`IGCanonicalLabeledEdgeList` is useful in conjunction with DeleteDuplicatesBy.

Highlight all distinct $K_{3,3}$ subgraphs of a Queen graph:

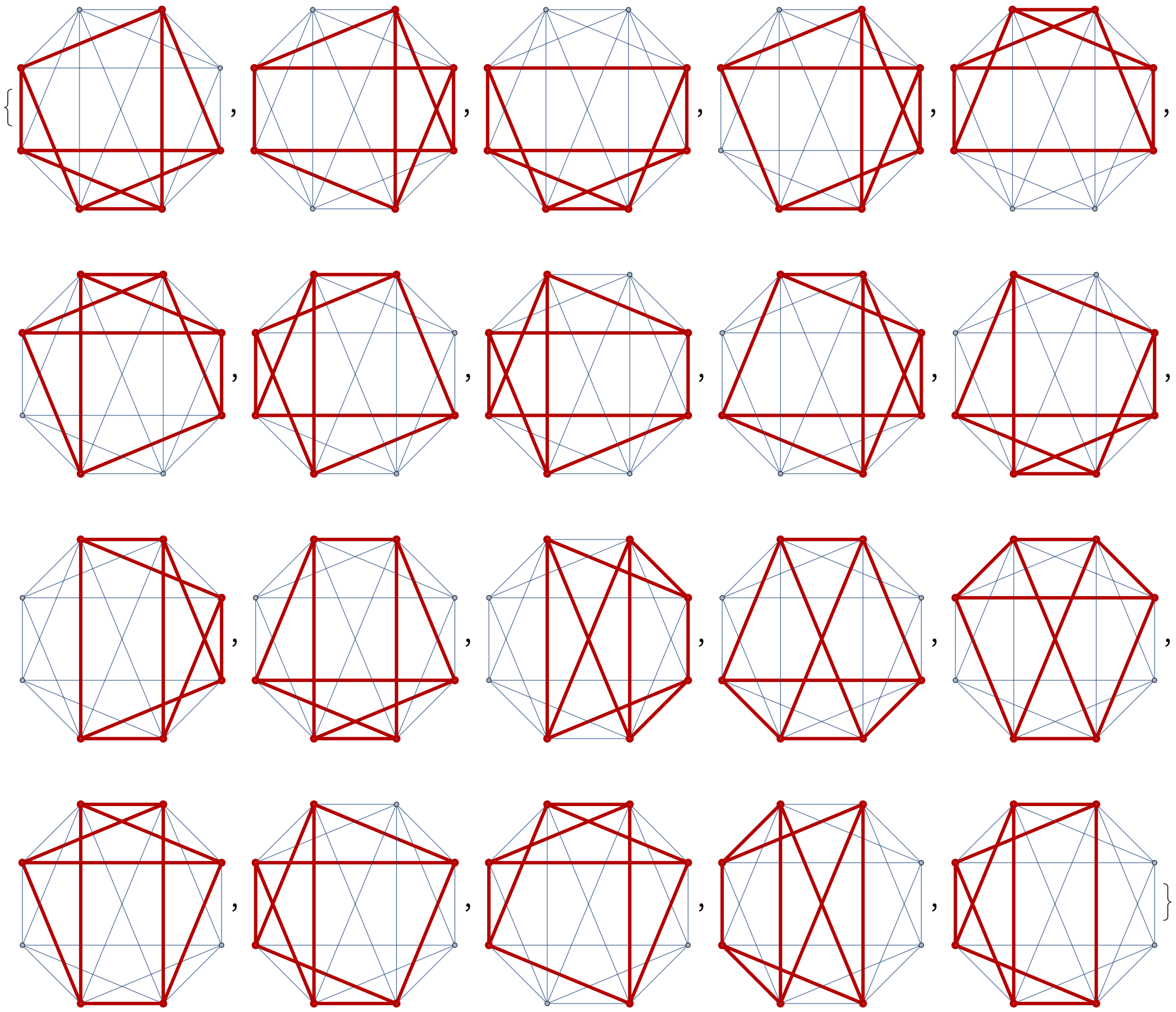
In[1650]:=

```
g = GraphData[{"Queen", {2, 4}}];
sg = CompleteGraph[{3, 3}];
```

In[1652]:=

```
HighlightGraph[g, Graph[#], GraphHighlightStyle → "Thick"] & /@ DeleteDuplicatesBy[
  IGCanonicalEdgeList@EdgeList[sg] /. IGLADFindSubisomorphisms[sg, g],
  IGCanonicalEdgeList
 ]
```

Out[1652]=



## IGAdjacentVerticesQ

In[1653]:=

```
? IGAdjacentVerticesQ
```

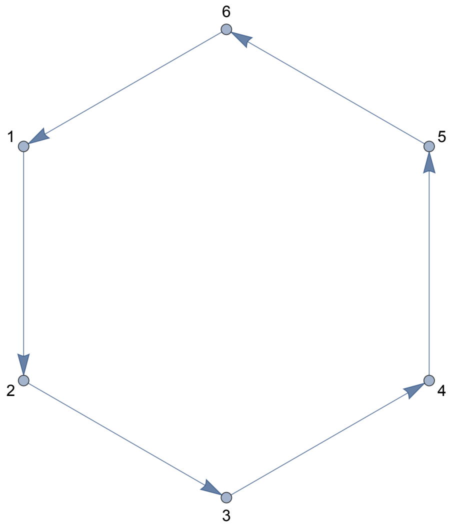
IGAdjacentVerticesQ[graph, {u, v}] tests if vertex v is adjacent to vertex u in graph.

`IGAdjacentVerticesQ[graph, {u, v}]` tests if there is an edge from u to v.

In[1654]:=

```
g = CycleGraph[6, DirectedEdges → True, VertexLabels → "Name"]
```

Out[1654]=




In[1655]:=

```
IGAdjacentVerticesQ[g, {1, 2}]
```

Out[1655]=

```
True
```

Edge directions are taken into account:

In[1656]:=

```
IGAdjacentVerticesQ[g, {2, 1}]
```

Out[1656]=

```
False
```

The following vertices are not adjacent:

In[1657]:=

```
IGAdjacentVerticesQ[g, {1, 3}]
```

Out[1657]=

```
False
```

Vertices that are not part of the graph are allowed. They are considered not to be adjacent:

In[1658]:=

```
IGAdjacentVerticesQ[g, {"x", 2}]
```

Out[1658]=

```
False
```

## IGPartitionsToMembership and IGMembershipToPartitions

In[1659]:=

```
? IGPartitionsToMembership
```

IGPartitionsToMembership[elements, partitions] computes a membership vector for the given partitioning of elements.
IGPartitionsToMembership[graph, partitions] computes a membership vector for the given partitioning of graph's vertices.
IGPartitionsToMembership[elements] is an operator that can be applied to partitions.

In[1660]:=

```
? IGMembershipToPartitions
```

IGMembershipToPartitions[elements, membership]
computes a partitioning of elements based on the given membership vector.
IGMembershipToPartitions[graph, membership]
computes a partitioning graph's vertices based on the given membership vector.
IGMembershipToPartitions[elements] is an operator that can be applied to membership vectors.

A partitioning of a set of elements can be represented in multiple ways. One way is to list the members of each partition. Another is to annotate each element with the index of the partition it belongs to, i.e. construct a “membership vector”. These functions convert between these representations.

IGraph/M generally uses disjoint subsets to represents partitions. Membership vectors are useful when storing the membership information as vertex attributes, or when exchanging data with other interfaces of igraph.

In[1661]:=

```
IGPartitionsToMembership[{a, b, c, d}, {{a, c}, {d, b}}]
```

Out[1661]=

```
{1, 2, 1, 2}
```

In[1662]:=

```
IGMembershipToPartitions[{a, b, c, d}, {1, 2, 1, 2}]
```

Out[1662]=

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

If the given partitions do not cover the element set, the missing elements will be marked with `0` in the membership vector.

In[1663]:=

```
IGPartitionsToMembership[{a, b, c, d}, {{a}, {b, d}}]
```

Out[1663]=

```
{1, 2, 0, 2}
```

The following graph has the type of nodes encoded as a vertex attribute.

In[1664]:=

```
g = ExampleData[{"NetworkGraph", "BipartiteDiseasomeNetwork"}];
IGVertexProp["Type"][g] // Short
```

Out[1665]//Short=

```
{Disease, Disease, Disease, Disease, Disease, <<3052>> , Entrez, Entrez, Entrez, Entrez}
```

Let us extract the attribute values as a vector and construct the two vertex partitions.

In[1666]:=

```
parts = IGMembershipToPartitions[g, IGVertexProp["Type"][g]];
```

Verify that the graph is bipartite according to this partitioning.

In[1667]:=

```
IGBipartiteQ[g, parts]
```

Out[1667]=

```
True
```

Annotate the vertices of a bipartite graph with their computed membership value.

In[1668]:=

```
g = IGBipartiteGameGNM[5, 6, 14, VertexSize → Large];
```

In[1669]:=

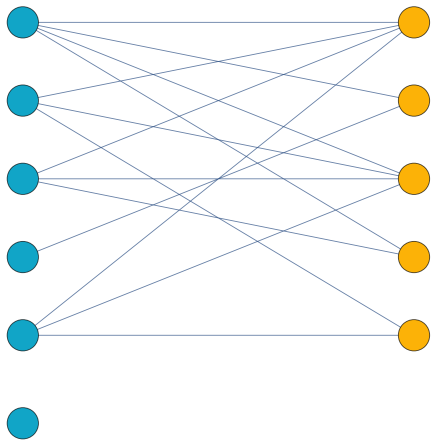
```
g = IGVertexMap[
   # &,
   "membership" → IGPartitionsToMembership[VertexList[g]]@*IGBipartitePartitions,
   g
  ];
```

Colour the vertices accordingly.

In[1670]:=

```
IGVertexMap[ColorData[100], VertexStyle → IGVertexProp["membership"], g]
```

Out[1670]=

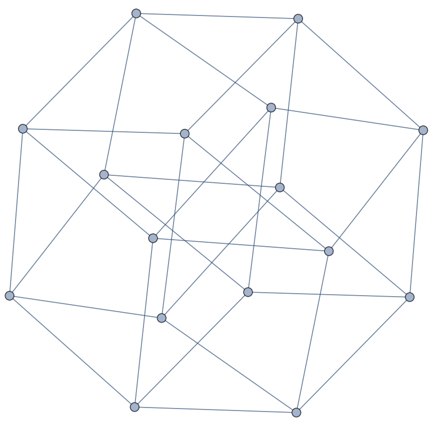


Visualize a vertex colouring using `HighlightGraph`.

In[1671]:=

```
g = IGSquareLattice[{2, 2, 2, 2}, "Periodic" → True]
```

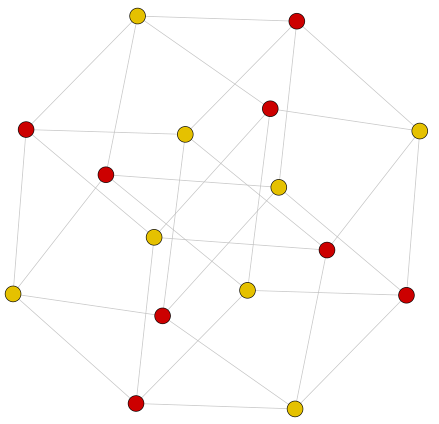
Out[1671]=



In[1672]:=

```
HighlightGraph[
 g, IGMembershipToPartitions[g, IGVertexColoring[g]],
 VertexSize → Medium, GraphHighlightStyle → "DehighlightGray"
]
```

Out[1672]=



## IGReorderVertices

In[1673]:=

```
? IGReorderVertices
```

IGReorderVertices[vertices, graph] reorders the vertices of graph according to the given vertex vector. Graph properties are preserved.

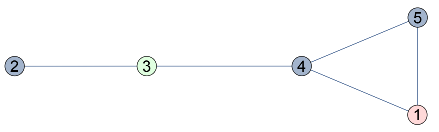
`IGReorderVertices` changes the order in which graph vertices are stored. The graph itself is not modified, only its representation. The ordering of vertices affects how several of *Mathematica*'s graph processing functions work.

Let us use a styled graph for illustration, to demonstrate that graph properties are preserved.

In[1674]:=

```
g1 = RandomGraph[{5, 5}, VertexStyle → {1 → LightRed, 3 → LightGreen},
  VertexSize → Medium, VertexLabels → Placed[Automatic, Center]]
```

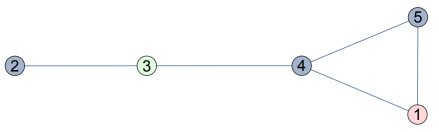
Out[1674]=




In[1675]:=

```
g2 = IGReorderVertices[{5, 4, 3, 2, 1}, g1]
```

Out[1675]=




In[1676]:=

```
VertexList /@ {g1, g2}
```

Out[1676]=

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

In[1677]:=

```
IGIsomorphicQ[g1, g2]
```

Out[1677]=

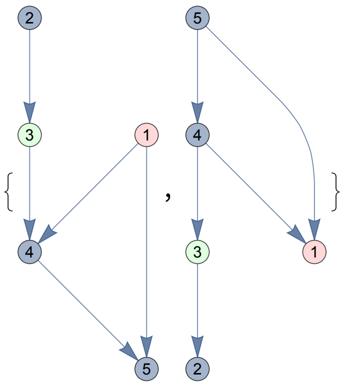
```
True
```

The result of certain operations, such as `DirectedGraph[..., "Acyclic"]` or `AdjacencyMatrix`, depends on the vertex ordering.

In[1678]:=

```
DirectedGraph[#, "Acyclic"] & /@ {g1, g2}
```

Out[1678]=

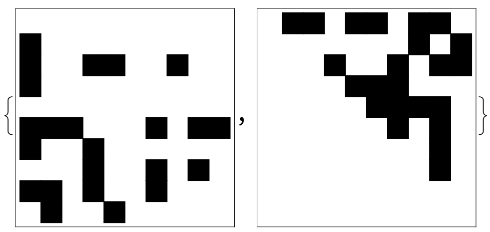



Order the vertices of a directed acyclic graph so that its adjacency matrix is upper triangular.

In[1679]:=

```
g = RandomGraph[{10, 30}, DirectedEdges → True];
g = EdgeDelete[g, IGFeedbackArcSet[g]];
ArrayPlot /@ AdjacencyMatrix /@ {g, IGReorderVertices[TopologicalSort[g], g]}
```

Out[1681]=

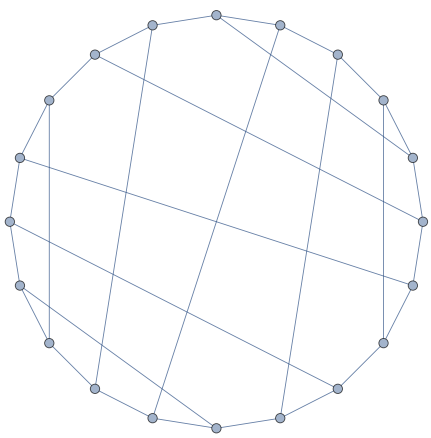


Visualize a graph so that a Hamiltonian cycle is on a circle.

In[1682]:=

```
g = GraphData["DodecahedralGraph"];
IGLayoutCircle@IGReorderVertices[FindHamiltonianCycle[g][[1, All, 1]], g]
```

Out[1683]=

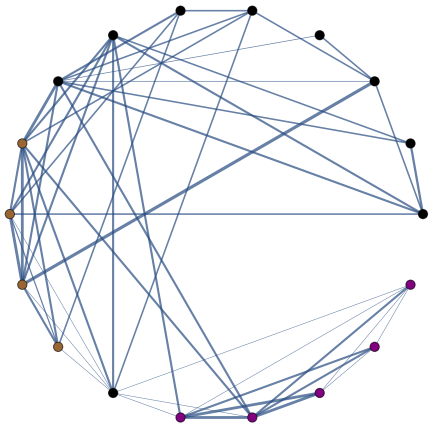


Change the order how graph vertices are drawn in a circular layout without discarding any styling or other properties.

In[1684]:=

```
g = IGLayoutCircle[
   ExampleData[{"NetworkGraph", "EastAfricaEmbassyAttacks"}],
   VertexLabels → (_ → Placed["Name", Tooltip])
  ]
```

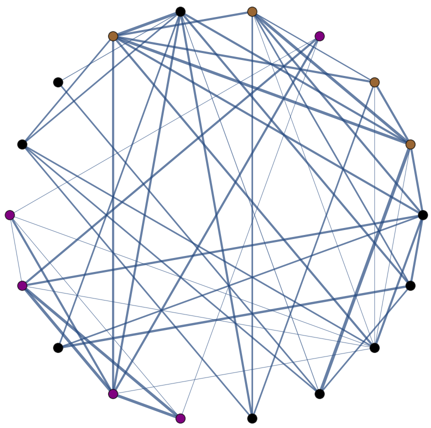
Out[1684]=



In[1685]:=

```
IGLayoutCircle@IGReorderVertices[RandomSample@VertexList[g], g]
```

Out[1685]=

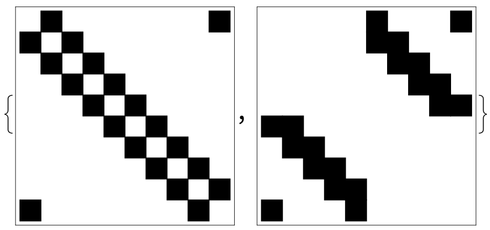


Reorder the vertices of a bipartite graph to make the bipartite structure explicit in its adjacency matrix. Note that if the goal is simply visualizing the adjacency matrix, `IGAdjacencyMatrixPlot` can be used instead.

In[1686]:=

```
g = CycleGraph[10];
ArrayPlot /@ AdjacencyMatrix /@ {g, IGReorderVertices[Flatten@IGBipartitePartitions[g], g]}
```

Out[1687]=

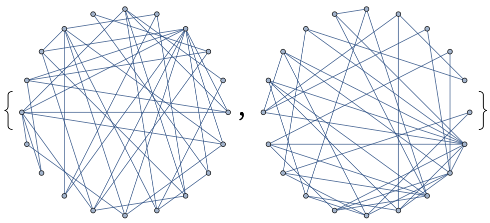


Order the vertices of a graph by increasing degree.

In[1688]:=

```
g = RandomGraph[{20, 40}];
{IGLayoutCircle[g],
 IGLayoutCircle@IGReorderVertices[VertexList[g][[Ordering@VertexDegree[g]]], g]}
```

Out[1689]=



## IGAdjacencyList

In[1690]:=

```
? IGAdjacencyList
```

IGAdjacencyList[graph] gives the adjacency list of graph as an association.
IGAdjacencyList[graph, "In"] gives the adjacency list of the reverse of a directed graph.
IGAdjacencyList[graph, "All"] considers both incoming and outgoing edges.

`IGAdjacencyList` returns the adjacency list of a graph as an association. This is often a more useful format than what the built-in `AdjacencyList` provides.

In[1691]:=

```
IGAdjacencyList[Graph[{1 ⟷ 2}]]
```

Out[1691]=

```
<|1 → {2}, 2 → {1}|>
```

For directed graphs, only outgoing edges are considered when building the adjacency list. In contrast, the built-in `AdjacencyList` ignores edge directions.

In[1692]:=

```
IGAdjacencyList[Graph[{1 → 2}]]
```

Out[1692]=

```
<|1 → {2}, 2 → {}|>
```

In[1693]:=

```
AdjacencyList[Graph[{1 → 2}]]
```

Out[1693]=

```
{{2}, {1}}
```

Consider incoming edges instead.

In[1694]:=

```
IGAdjacencyList[Graph[{1 → 2}], "In"]
```

Out[1694]=

```
<|1 → {}, 2 → {1}|>
```

Consider both incoming and outgoing edges.

In[1695]:=

```
IGAdjacencyList[Graph[{1 → 2}], "All"]
```

Out[1695]=

```
<|1 → {2}, 2 → {1}|>
```

With this option, reciprocal edges are considered individually in directed graphs.

In[1696]:=
```
IGAdjacencyList[Graph[{1 → 2, 2 → 1}], "All"]
```
Out[1696]=
```
<|1 → {2, 2}, 2 → {1, 1}|>
```

Multi-edges and self-loops are supported. In contrast, the built-in `AdjacencyList` ignores them.

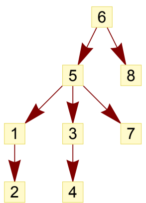
In[1697]:=
```
IGAdjacencyList[Graph[{1 → 2, 1 → 2, 2 → 2, 2 → 2}]]
```
Out[1697]=
```
<|1 → {2, 2}, 2 → {2, 2}|>
```

In[1698]:=
```
AdjacencyList[Graph[{1 → 2, 1 → 2, 2 → 2}]]
```
Out[1698]=
```
{{2}, {1, 2}}
```

Self-loops are traversed in only one direction in undirected graphs. Thus the result of the below is not `<|1 → {1, 1}|>` but simply `<|1 → {1}|>`. This is consistent with `AdjacencyMatrix`, but not with `VertexDegree`.

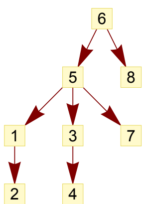
In[1699]:=
```
IGAdjacencyList[Graph[{1 ⟷ 1}]]
```
Out[1699]=
```
<|1 → {1}|>
```

`IGAdjacencyList` can be used to find the parent of each node in a rooted tree. The root itself will have no parent.

In[1700]:=
```
IGAdjacencyList[
```


```
, "In"]
```
Out[1700]=
```
<|1 → {5}, 2 → {1}, 3 → {5}, 4 → {3}, 5 → {6}, 6 → {}, 7 → {5}, 8 → {6}|>
```

Find the children of each node.

In[1701]:=
```
IGAdjacencyList[
```


```
, "Out"]
```
Out[1701]=
```
<|1 → {2}, 2 → {}, 3 → {4}, 4 → {}, 5 → {1, 3, 7}, 6 → {5, 8}, 7 → {}, 8 → {}|>
```

## IGAdjacencyGraph

In[1702]:=
```
? IGAdjacencyGraph
```

IGAdjacencyGraph[matrix] creates a graph from the given adjacency matrix.
IGAdjacencyGraph[vertices, matrix] creates a graph with the given vertices from an adjacency matrix.
IGAdjacencyGraph[adjList] creates a graph from an association representing an adjacency list.

`IGAdjacencyGraph` can convert an adjacency matrix or an adjacency list representation of a graph into a `Graph` expression. When given a matrix, it behaves equivalently to the built-in function `AdjacencyGraph`.

The available options are:

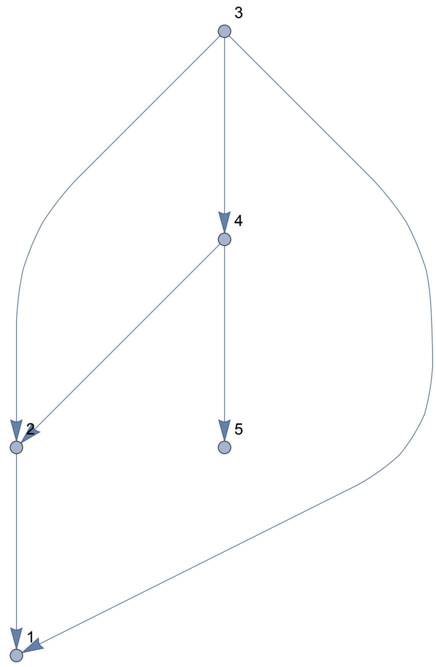
- `DirectedEdges → True` and `DirectedEdges → False` create a directed or undirected graph, respectively. The default setting is `DirectedEdges → Automatic`, which creates an undirected graph when this is consistent with the given adjacency matrix or adjacency list.

Compute the adjacency list of a graph, then convert it back to a `Graph` expression.

In[1703]:=

```
g = RandomGraph[{5, 6}, DirectedEdges → True, VertexLabels → "Name"]
```

Out[1703]=




In[1704]:=

```
IGAdjacencyList[g]
```

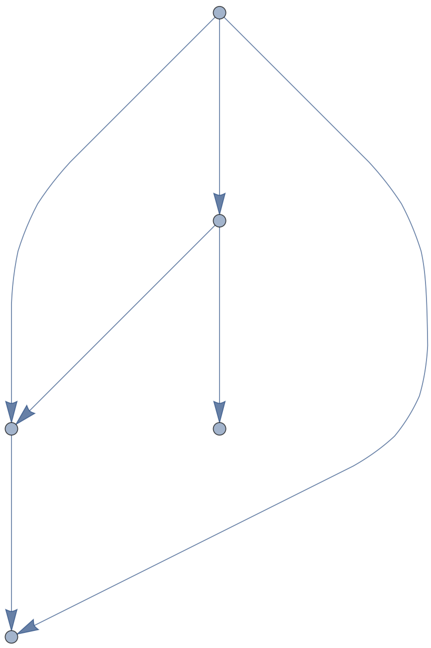
Out[1704]=

```
<|1 → {}, 2 → {1}, 3 → {1, 2, 4}, 4 → {2, 5}, 5 → {}|>
```

In[1705]:=

```
IGAdjacencyGraph[%]
```

Out[1705]=



The representation of combinatorial embeddings used by IGraph/M is also a valid adjacency list.

In[1706]:=

```
IGPlanarEmbedding@CompleteGraph[4]
```

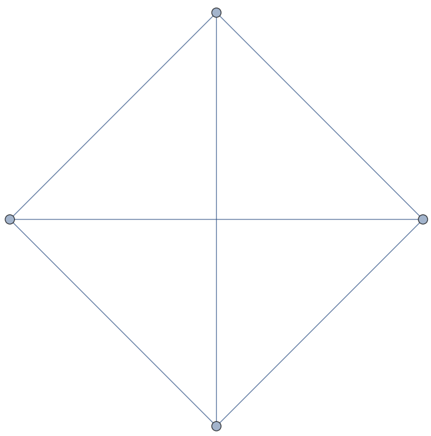
Out[1706]=

```
<|1 → {2, 3, 4}, 2 → {1, 4, 3}, 3 → {2, 4, 1}, 4 → {3, 2, 1}|>
```

In[1707]:=

```
IGAdjacencyGraph[%]
```

Out[1707]=



## IGVertexAssociate

In[1708]:=

```
? IGVertexAssociate
```

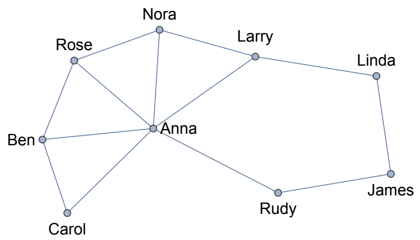
IGVertexAssociate[fun][graph] associates the result of fun[graph] with the vertices of graph.
IGVertexAssociate[fun][graph, vertices] associates the result of fun[graph, vertices] with vertices.

`IGVertexAssociate[fun]` is an operator that, when applied to `graph`, will associate the result of `fun[graph]` with each vertex.

In *Mathematica*, functions that compute a value for each vertex always return a list, with the values ordered to correspond to the `VertexList` of the graph. In many situations, it is more convenient to use an association where the keys are the vertex names. If `fun` is a function that computes a vertex property and gives the result as a list, the operator `IGVertexAssociation[fun]` will give an association instead.

Get the betweenness of a vertex by name:

In[1709]:=
```
net = ExampleData[{"NetworkGraph", "Friendship"}]
```
Out[1709]=




In[1710]:=
```
betw = IGVertexAssociate[IGBetweenness][net]
```
Out[1710]=
```
<|Anna → 15.5, Rose → 0.5, Nora → 1., Ben → 0.5,
 Larry → 5.5, Carol → 0., Rudy → 4.5, Linda → 1.5, James → 1.|>
```

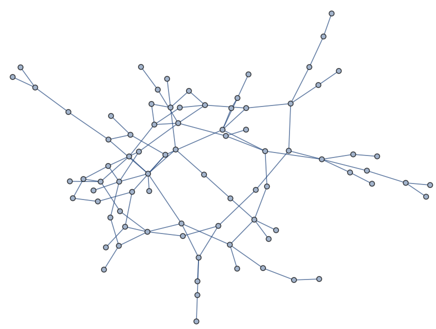
In[1711]:=
```
betw["Larry"]
```
Out[1711]=
```
5.5
```

IGraph/M has many functions which can be restricted to compute values for only a subset of vertices. These use the syntax `fun[graph, vertices]`. If `fun` supports this syntax, then `IGVertexAssociate[fun]` also takes a vertex list as its second argument.

In[1712]:=
```
IGVertexAssociate[IGEccentricity][net, {"Anna", "James", "Rudy"}]
```
Out[1712]=
```
<|Anna → 2, James → 3, Rudy → 2|>
```

Smoothen away the degree-2 vertices of a graph while retaining the coordinates of each vertex:

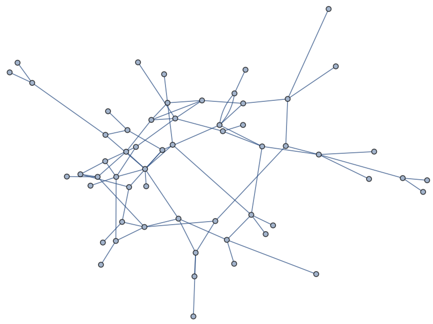
In[1713]:=
```
g = IGGiantComponent@RandomGraph[{100, 100}]
```
Out[1713]=



In[1714]:=

```
g2 = IGSmoothen[g] //
  IGVertexMap[IGVertexAssociate[GraphEmbedding][g], VertexCoordinates → VertexList]
```

Out[1714]=

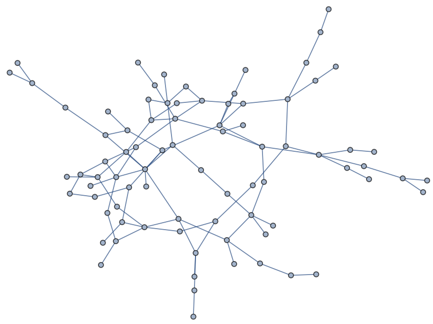


Compare the smoothened graph with the original in a flip view:

In[1715]:=

```
FlipView[{g, g2}]
```

Out[1715]=



Extract a vertex property as an association:

In[1716]:=

```
IGVertexAssociate[IGVertexProp["Group"]]@
 ExampleData[{"NetworkGraph", "EastAfricaEmbassyAttacks"}]
```

Out[1716]=

```
⟨|Osama → Planners, Salim → Planners, Ali → Planners, Abouhlaima → Planners,
 Kherchtou → Planners, Fawwaz → Planners, Abdullah → Planners, Hage → Planners,
 Odeh → Nairobi Cell, Owhali → Nairobi Cell, Fazul → Nairobi Cell, Azzam → Nairobi Cell,
 Atwah → Planners, Fahad → Dar es Salaam Cell, Fadhil → Dar es Salaam Cell,
 Khalfan → Dar es Salaam Cell, Ghailani → Dar es Salaam Cell, Awad → Dar es Salaam Cell|⟩
```

## IGTryUntil

In[1717]:=

```
? IGTryUntil
```

IGTryUntil[cond][expr] repeatedly evaluates expr until cond[expr] is True.
IGTryUntil[cond, max][expr] evaluates expr at most max times and returns $Failed if cond[expr] was never True.

`IGTryUntil` repeatedly evaluates an expression until the result of the evaluation satisfies a condition. It is particularly useful for concisely implementing rejection sampling.

Choose 10 distinct random primes not greater than 100:

In[1718]:=

```
IGTryUntil[DuplicateFreeQ][RandomPrime[100, 10]]
```

Out[1718]=

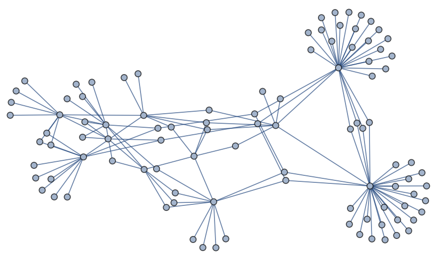
```
{53, 61, 73, 41, 43, 19, 17, 67, 37, 11}
```

Create a power-law distributed degree sequence and build a corresponding graph:

In[1719]:=

```
IGRealizeDegreeSequence[
 IGTryUntil[IGGraphicalQ]@RandomVariate[ZipfDistribution[1], 100]
]
```

Out[1719]=

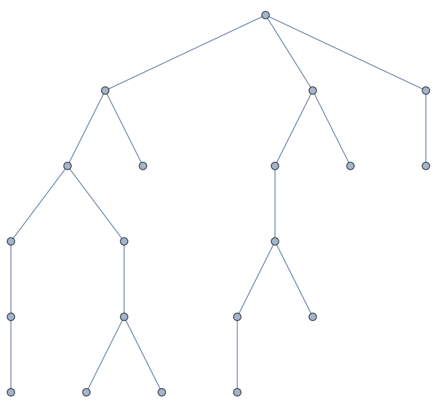


Generate a random tree (a connected graph) with a given degree sequence using the configuration model:

In[1720]:=

```
ds = {3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1};
IGTryUntil[IGConnectedQ]@IGDegreeSequenceGame[ds, Method → "ConfigurationModelSimple"]
```

Out[1721]=



Some result will occur very infrequently or not at all, so it is useful to limit the number of trials. The following attempts to generate a random non-connected cubic graph on 50 vertices, and simply returns `$Failed` if it does not succeed after 100 tries.

In[1722]:=

```
IGTryUntil[Not@*IGConnectedQ, 100][
 IGDegreeSequenceGame[
  ConstantArray[3, 50],
  Method → "ConfigurationModelSimple"
 ]
]
```

Out[1722]=

```
$Failed
```

# Built-in data

## Graph data

The `IGData[]` function provides access to various useful datasets. In particular, it can list small directed graphs ordered based on their `IGIsoclass[]`, i.e. the same order that motif counting functions use.

In[1723]:=

```
? IGData
```

IGData[] returns a list of available items.
IGData[item] returns the requested item.

List the available datasets:

In[1724]:=

```
IGData[]
```

Out[1724]=

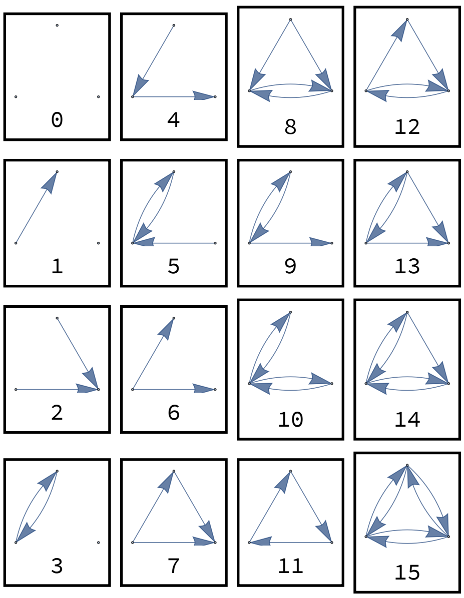
```
{{AllDirectedGraphs, 2}, {AllDirectedGraphs, 3}, {AllDirectedGraphs, 4},
 {AllUndirectedGraphs, 2}, {AllUndirectedGraphs, 3}, {AllUndirectedGraphs, 4},
 {AllUndirectedGraphs, 5}, {AllUndirectedGraphs, 6}, MANTriadLabels}
```

These are all size 3 directed graphs:

In[1725]:=

```
IGData[{"AllDirectedGraphs", 3}] //
  Map[Framed@Labeled[Graph[#, ImageSize → 50], IGIsoclass[#]] &] // Multicolumn
```

Out[1725]=



"MANTriadLabels" refers to the mutual, asymmetric, null labelling of triads used by IGTriadCensus[]. Each label is mapped to the corresponding graph, ordered based on their IGIsoclass. This is useful for converting the output of IGTriadCensus[] to a format compatible with IGMotifs[].

In[1726]:=

```
g = RandomGraph[{20, 50}, DirectedEdges → True];
{IGMotifs[g, 3], Lookup[IGTriadCensus[g], Keys@IGData["MANTriadLabels"]]} // Grid
```

Out[1727]=

```
Indeterminate Indeterminate 38 Indeterminate 72 6 50 10 1 6 0 1 1 1 0 0
     476           457       38      21       72 6 50 10 1 6 0 1 1 1 0 0
```

The IGGraphAtlas function provides access to the graphs listed in An Atlas of Graphs by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

In[1728]:=

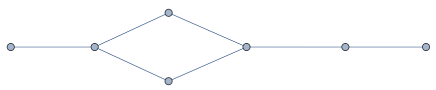
```
? IGGraphAtlas
```

IGGraphAtlas[n] gives graph number n from An Atlas of Graphs by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998. This function is provided for convenience; if you are looking for a specific named graph, use the builtin GraphData function.

In[1729]:=

```
IGGraphAtlas[341]
```

Out[1729]=



Finally, remember that *Mathematica* itself comes with a large database of graphs and their properties, accessible through GraphData.

## Lattice data

The IGLatticeMesh function includes a set of pre-defined two-dimensional lattice structures. Evaluate IGLatticeMesh[] to get the list of available lattices.

The data used by IGMeshGraph was sourced from Wolfram|Alpha and *Mathematica*'s curated data system in April 2018.

In[1730]:=

```
? IGLatticeMesh
```

IGLatticeMesh[type] creates a mesh of the lattice of the specified type.
IGLatticeMesh[type, {m, n}] creates a lattice of n by m unit cells.
IGLatticeMesh[type, region] creates a lattice from the points that fall within region.
IGLatticeMesh[] gives a list of available lattice types.

In[1731]:=

```
IGLatticeMesh[]
```

Out[1731]=

```
{Square, Hexagonal, Triangular, Trihexagonal, SmallRhombitrihexagonal, TruncatedSquare,
 SnubSquare, TruncatedHexagonal, ElongatedTriangular, GreatRhombitrihexagonal, SnubHexagonal,
 Rhombille, DeltoidalTrihexagonal, TetrakisSquare, CairoPentagonal, TriakisTriangular,
 PrismaticPentagonal, BisectedHexagonal, FloretPentagonal, DellaRobbiaWeave, Portugal,
 StackBond, Herringbone, Basketweave, PersianHexagonalWeave, Hopscotch, StretcherBond,
 Pinwheel, BrickworkSquare, Chickenwire, Corridor, CorridorHorizontal, Brickweave,
 Trellis, HeeschIsohedral, PPentomino, Chevron, Shingle, Zigzag, Kite, FalseCubic,
 TrihexAndHex, GlideReflection, PentagonType1, PentagonType2, PentagonType3, PentagonType4,
 PentagonType5, PentagonType6, PentagonType7, PentagonType8, PentagonType9, PentagonType10,
 PentagonType11, PentagonType12, PentagonType13, PentagonType14, PentagonType15}
```

# IGraph/M system functions

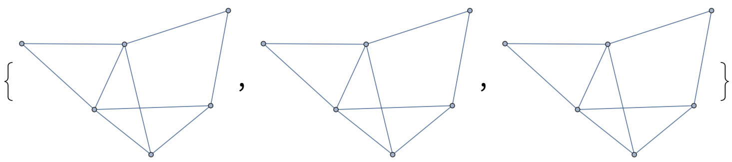
## The random number generator

IGraph/M makes use of *Mathematica*'s own random number generator by default, thus functions like SeedRandom and BlockRandom have the expected effect.

In[1732]:=

```
SeedRandom[137];
Table[BlockRandom@IGErdosRenyiGameGNM[6, 9], {3}]
```

Out[1733]=

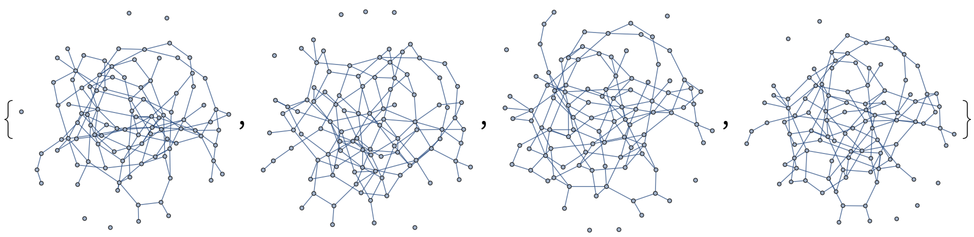


BlockRandom is useful for example to get consistent graph layouts without affecting subsequent uses of the random number generator.

In[1734]:=

```
g = RandomGraph[{100, 150}];
Table[IGLayoutFruchtermanReingold[g], {4}]
```

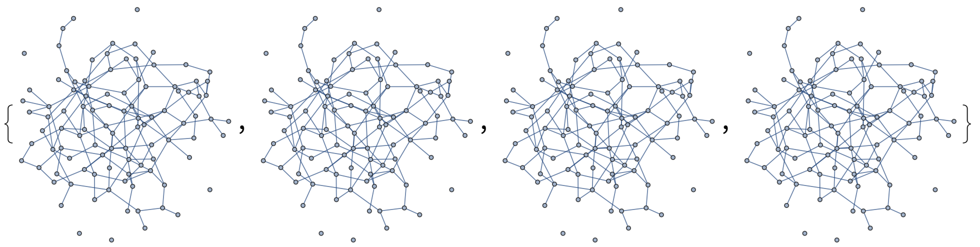
Out[1735]=



In[1736]:=

```
Table[BlockRandom[SeedRandom[1234]; IGLayoutFruchtermanReingold[g]], {4}]
```

Out[1736]=



IGraph/M can be configured to either use *Mathematica*'s built-in generator, or the default generator of the igraph C library. The default generator of igraph will perform better, but it does not react to `BlockRandom` and must be seeded with `IGSeedRandom` (not with `SeedRandom`).

Benchmark `IGRandomWalk` when using *Mathematica*'s random number generator:

In[1737]:=

```
g = IGGiantComponent@RandomGraph[{1000, 2000}];
```

In[1738]:=

```
IGRandomWalk[g, 1, 10 000 000]; // RepeatedTiming
```

Out[1738]=

```
{0.630358, Null}
```

Benchmark it with igraph's default generator:

In[1739]:=

```
IGSeedRandom[Method → "igraph"]
IGRandomWalk[g, 1, 10 000 000]; // RepeatedTiming
```

Out[1740]=

```
{0.388662, Null}
```

Set the generator back to *Mathematica*'s:

In[1741]:=

```
IGSeedRandom[Method → "Mathematica"]
```

### IGSeedRandom

In[1742]:=

```
? IGSeedRandom
```

IGSeedRandom[seed] seeds the current random number generator.
IGSeedRandom[Method -> type] sets the current random number generator. Valid types are "Mathematica" and "igraph".

Available `Method` option values are:

- `"Mathematica"` uses Mathematica's built-in random number generator. With this choice, functions like `SeedRandom` and `BlockRandom` will IGraph/M functions as expected. Performance is not as good as with the `"igraph"` generator
- `"igraph"` uses the core igraph C library's random number generator. `SeedRandom` and `BlockRandom` have no effect on this generator. Seeding can be done with `IGSeedRandom`. Performance is better than with the `"Mathematica"` generator.

## Progress reporting

**Experimental:** This is experimental functionality that may change in the future.

Some igraph functions can report their progress while working. IGraph/M contains experimental functionality that exposes igraph's progress reports. This functionality may change without notice in the future.

In[1743]:=

```
? IGraphM`Progress`*
```

▼ IGraphM`Progress`

| | | | |
|---|---|---|---|
| Indicator | Message | Percent | SetReporting |

Show the progress indicator.

In[1744]:=

```
IGraphM`Progress`Indicator[]
```

Out[1744]=

Progress reporting has a performance cost, therefore it is disabled by default. To enable it, use:

In[1745]:=

```
IGraphM`Progress`SetReporting[True]
```

When a computation that supports progress reporting is running, the indicator will show the status.

In[1746]:=

```
compute[] := IGCommunitiesGreedy@
   IGStochasticBlockModelGame[0.02 IdentityMatrix[10] + 0.005, ConstantArray[800, 10]];

compute[];
```

By default, the progress indicator is updated only if progress has increased by at least 1%. In other words, the reporting granularity is 1%. The lower the granularity value, the higher the performance impact of reporting.

Change the reporting granularity to 10%.

In[1748]:=

```
IGraphM`Progress`SetReporting[True, "Granularity" → 10]
```

Follow the progress by dynamically showing the value of internal progress variables:

In[1749]:=

```
Dynamic@{IGraphM`Progress`Message, IGraphM`Progress`Percent}

compute[];
```

Out[1749]=

```
{, 0.}
```

Disable progress reporting and set the granularity to its default values (in case it gets enabled again later).

In[1751]:=

```
IGraphM`Progress`SetReporting[False, "Granularity" → Automatic]
```

## Library version

The following symbols and functions can be used to retrieve the IGraph/M version.

In[1752]:=

```
? IGVersion
```

IGVersion[] returns the IGraph/M version along with the version of the igraph library in use.

In[1753]:=

```
IGVersion[]
```

Out[1753]=

```
IGraph/M 0.6.5 (December 21, 2022)
igraph 0.9.10-23-g5635203bd (Dec 21 2022)
Mac OS X x86 (64-bit)
```

In[1754]:=

```
? IGraphM`Information`$Version
```

IGraphM`Information`$Version is a string that gives the version of the currently loaded IGraph/M package.

In[1755]:=

```
IGraphM`Information`$Version
```

Out[1755]=

```
0.6.5 (December 21, 2022)
```

# Support and troubleshooting

If you need help with using this package, the following support options are available:

- Post on the igraph discussion forum and tag the post as `Mathematica`.
- Post on the Mathematica StackExchange and tag the post as `igraphm`.

If you find a problem with IGraph/M or its documentation, please report it through the GitHub issue tracker or the igraph discussion forum. Always include the output of the `GetInfo[]` function with problem reports.

In[1756]:=

```
? IGraphM`Developer`GetInfo
```

IGraphM`Developer`GetInfo[] returns useful information about
IGraph/M and the system it is running on, for debugging and troubleshooting purposes.

# Acknowledgements

Most functions in IGraph/M are based on the igraph C library, originally written by Gábor Csárdi and Tamás Nepusz. To cite the igraph C library in publications, see “Citing igraph” in the igraph Reference Manual. Website: https://igraph.org/

Some functions, in particular in the area of planar graphs, use the LEMON graph library. Website: https://lemon.cs.elte.hu/

Some proximity graph functions make use of the nanoflann library. Website: https://github.com/jlblancoc/nanoflann

IGraph/M was developed with the Wolfram Language Plugin for IntelliJ IDEA by Patrick Scheibe. Without the help of this IDE, it would have been difficult to manage the complexity of this package. Website: http://wlplugin.halirutan.de/

The web version of the documentation is prepared with the M2MD package by Kuba Podkalicki. Website: https://github.com/kubaPod/M2MD/

The help of the Mathematica StackExchange community was invaluable while developing this package.

People who have contributed to IGraph/M:

- Szabolcs Horvát (main author and maintainer)
- Henrik Schumacher (help with mesh-graph conversion and proximity graph functions)
- Juho Lauri (advice with the implementation of graph colouring functions)
- Kuba Podkalicki (implementation of `IGGraphEditor[]`)

**To cite IGraph/M in a publication, please refer to:**

- Sz. Horvát, J. Podkalicki, G. Csárdi, T. Nepusz, V. Traag, F. Zanini, D. Noom, *IGraph/M: graph theory and network analysis for Mathematica*, preprint (2022), doi:10.48550/arXiv.2209.09145
- IGraphM/ on Zenodo, doi:10.5281/zenodo.1134932

---

# License