

Wolfram *Mathematica*® Tutorial Collection

GRAPH DRAWING



For use with Wolfram Mathematica® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

©2008 Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.

Wolfram Research is the holder of the copyright to the Wolfram Mathematica software system ("Software") described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, "look and feel," programming language, and compilation of command names. Use of the Software unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.

Mathematica, MathLink, and MathSource are registered trademarks of Wolfram Research, Inc. J/Link, MathLM, .NET/Link, and webMathematica are trademarks of Wolfram Research, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh is a registered trademark of Apple Computer, Inc. All other trademarks used herein are the property of their respective owners. Mathematica is not associated with Mathematica Policy Research, Inc.

Contents

Introduction	1
Graph Theory Notations	2
Input Formats	3
Graph Drawing Algorithms	6
Selecting the Appropriate Graph Drawing Function	11
References	12
General Graph Drawing	14
Options for GraphPlot and GraphPlot3D	16
Common Suboptions of All Methods	42
Common Suboptions of the "SpringEmbedding" and "SpringElectricalEmbedding" Methods	43
Method Suboptions of the "SpringElectricalEmbedding" Method	49
Method Suboption of "HighDimensionalEmbedding"	50
Advanced Topics	51
Example Gallery	57
References	66
Hierarchical Drawing of Directed Graphs	67
Options for LayeredGraphPlot	68
Example Gallery	81
Tree Drawing	85
Options for TreePlot	87
Example Gallery	101

Introduction to Graph Drawing

Mathematica provides functions for the aesthetic drawing of graphs. Algorithms implemented include spring embedding, spring-electrical embedding, high-dimensional embedding, radial drawing, random embedding, circular embedding, and spiral embedding. In addition, algorithms for layered/hierarchical drawing of directed graphs as well as for the drawing of trees are available. These algorithms are implemented via four functions: `GraphPlot`, `GraphPlot3D`, `LayeredGraphPlot`, and `TreePlot`.

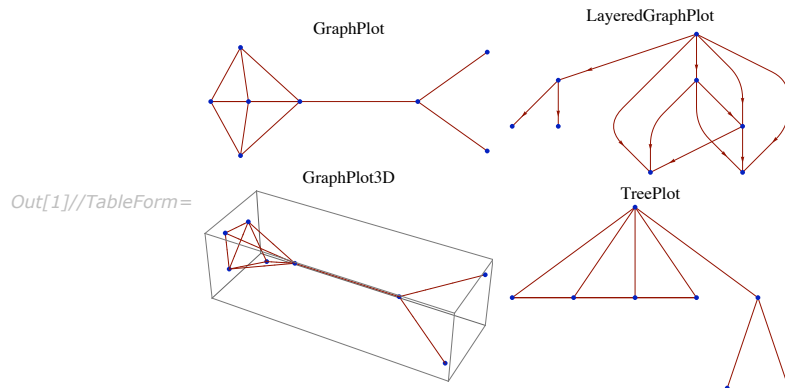
<code>GraphPlot</code>	generate a plot of a graph
<code>GraphPlot3D</code>	generate a 3D plot of a graph
<code>LayeredGraphPlot</code>	generate a layered plot of a graph
<code>TreePlot</code>	generate a tree plot of a graph

Functions for graph drawing.

`GraphPlot` and `GraphPlot3D` are suitable for straight line drawing of general graphs. `LayeredGraphPlot` attempts to draw the vertices of a graph in a series of layers; therefore it is most suitable for applications such as the drawing of flow charts. `TreePlot` is particularly useful for drawing trees or tree-like graphs. These functions are designed to work efficiently for very large graphs.

This shows a graph drawn using each of the four functions.

```
In[1]:= {Map[#[{4 → 2, 4 → 3, 5 → 2, 5 → 3, 5 → 4, 6 → 2, 6 → 3, 6 → 4, 6 → 5, 6 → 7, 7 → 8, 7 → 9},  
PlotRangePadding -> Automatic, ImageSize -> 180, PlotLabel -> #] &,  
{GraphPlot, GraphPlot3D}, {LayeredGraphPlot, TreePlot}], {-1}] // TableForm
```



In these functions, a graph is represented either by a list of rules of the form $\{v_{i_1} \rightarrow v_{j_1}, \dots\}$, where v_{i_1} and v_{j_1} are vertices, or by the adjacency matrix of the graph. Graphs in the *Combinatorica* package format are also supported.

Graph Theory Notations

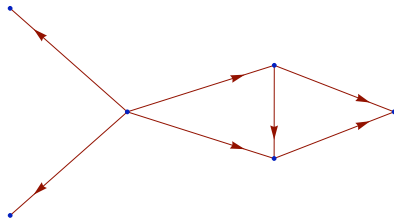
A graph $G = \{V, E\}$ consists of a set of vertices V (also called nodes) and a set of edges E . Two vertices u and v form an edge of the graph if $\{u, v\} \in E$.

If $\{u, v\} \in E$ implies that $\{v, u\} \in E$, then G is an undirected graph. Otherwise it is a directed graph. The former can be drawn using line segments, while the latter can be drawn with arrows. In an undirected graph, it is often convenient to denote that an edge exists between u and v with the notation $u \leftrightarrow v$.

For example, this is a directed graph.

```
In[2]:= GraphPlot[{4 -> 3, 5 -> 3, 5 -> 4, 6 -> 1, 6 -> 2, 6 -> 4, 6 -> 5},
  DirectedEdges -> True]
```

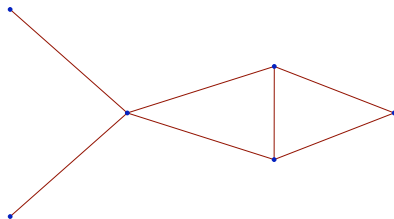
Out[2]=



Here is an undirected graph.

```
In[3]:= GraphPlot[{4 -> 3, 5 -> 3, 5 -> 4, 6 -> 1, 6 -> 2, 6 -> 4, 6 -> 5}]
```

Out[3]=

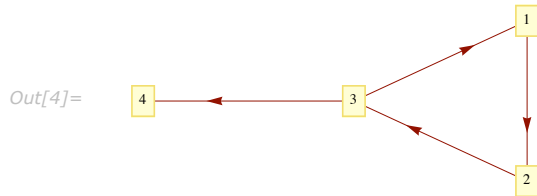


Input Formats

In *Mathematica*, graphs can be represented by one of the following three data structures. A graph can be represented by a list of rules.

For example, $\{1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1, 3 \rightarrow 4\}$ represents the following directed graph.

```
In[4]:= GraphPlot[{1 -> 2, 2 -> 3, 3 -> 1, 3 -> 4}, DirectedEdges -> True,
VertexLabeling -> True]
```



A graph can also be represented by its adjacency matrix. Let $G = \{V, E\}$ be a directed graph. Assuming that the vertices are indices from 1 to n , that is, $V = \{1, 2, \dots, n\}$, then the adjacency matrix of G is an $n \times n$ matrix, with entries $a_{ij} = 1$ if $\{i, j\} \in E$ and $a_{ij} = 0$ otherwise.

The following adjacency matrix represents the same directed graph.

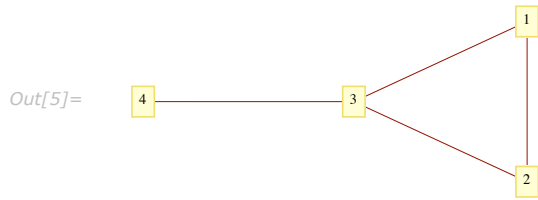
$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

An undirected graph, on the other hand, is represented by a symmetric adjacency matrix. The matrix entries $a_{ij} = a_{ji} = 1$ if $\{i, j\} \in E$ and $a_{ij} = 0$ otherwise.

This adjacency matrix represents the undirected graph that follows it.

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

```
In[5]:= GraphPlot[ $\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ , VertexLabeling → True]
```



Because of the zero entries in an adjacency matrix, it is often convenient to represent the matrix using a `SparseArray`.

The previous matrix can be written as the following sparse array.

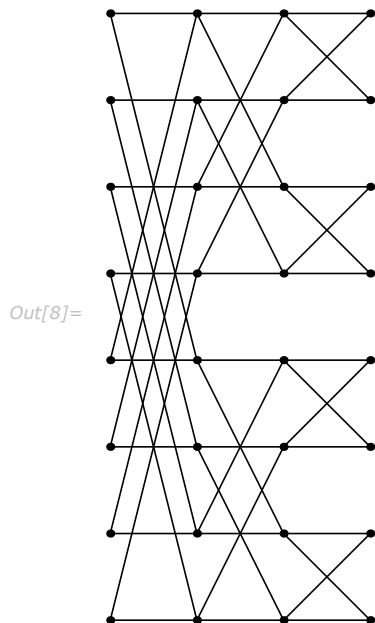
```
In[6]:= SparseArray[{{1, 2} → 1, {1, 3} → 1, {2, 1} → 1,
  {2, 3} → 1, {3, 1} → 1, {3, 2} → 1, {3, 4} → 1, {4, 3} → 1}, {4, 4}];
```

Finally, graphs in the *Combinatorica* package format are also supported.

This example creates a butterfly graph using *Combinatorica* and shows the layout *Combinatorica* assigned.

```
In[7]:= << Combinatorica`;
```

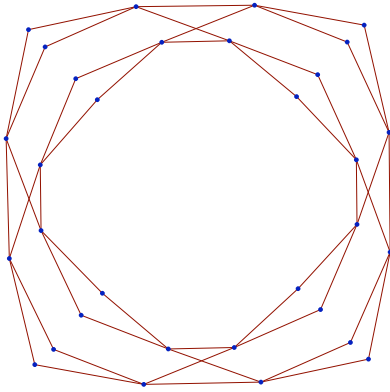
```
In[8]:= g = ButterflyGraph[3]; ShowGraph[g]
```



This draws the same graph using GraphPlot.

```
In[9]:= GraphPlot[g]
```

Out[9]=

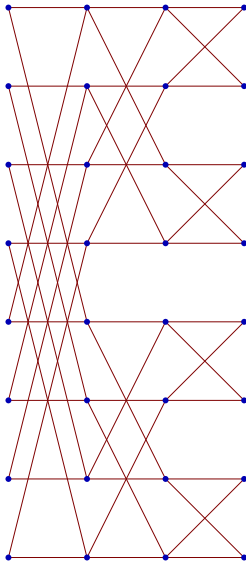


GraphPlot uses the algorithms described in the next section to lay out a graph. If GraphPlot is to be used for a graph in *Combinatorica* format, but the drawing assigned by *Combinatorica* is to be preserved, Method -> None can be specified.

Option Method -> None draws the graph using the layout from the *Combinatorica* package.

```
In[10]:= GraphPlot[g, Method -> None]
```

Out[10]=



Graph Drawing Algorithms

Graphs are often used to encapsulate the relationship between items. Graph drawing enables visualization of these relationships. The usefulness of the visual representation depends upon whether the drawing is aesthetic. While there are no strict criteria for aesthetic drawing, it is generally agreed that such a drawing has minimal edge crossing and even spacing between vertices. This problem has been studied extensively in the literature [1], and many approaches have been proposed. Two popular straight-edge drawing algorithms, the spring embedding and spring-electrical embedding, work by minimizing the energy of physical models of the graph. The high-dimensional embedding method, on the other hand, embeds a graph in high-dimensional space and then projects it back to two- or three-dimensional space. In addition, there are algorithms for drawing directed graphs in a hierarchical fashion, as well as for drawing trees. Random embedding, circular embedding, and spiral embedding do not utilize any connectivity information for laying out a graph, and therefore are not described any further here.

Spring Embedding

The spring embedding algorithm assigns force between each pair of nodes. When two nodes are too close together, a repelling force comes into effect. When two nodes are too far apart, they are subject to an attractive force. This scenario can be illustrated by linking the vertices with springs—hence the name "spring embedding."

This algorithm works by adding springs to all edges and adding looser springs to all vertex pairs that are not adjacent. Thus, in two dimensions, the total energy of the system is

$$\sum_{i=1}^{|V|-1} \sum_{j=i+1}^{|V|} k_{ij} \left(\|x_i - x_j\|^2 - l_{ij} \right)^2.$$

Here, x_i and x_j are the coordinate vectors of nodes i and j , and $\|x_i - x_j\|$ is the Euclidean distance between them. l_{ij} is the natural length of the spring between vertex i and vertex j , and can be chosen as the graph distance between i and j . The parameters $k_{ij} = R/l_{ij}^2$ are the strength of the springs, where R is a parameter representing the strength of the springs. $|V|$ is the number of vertices.

The layout of the graph vertices is calculated by minimizing this energy function. One way to minimize the energy function is by iteratively moving each of the vertices along the direction of the spring force until an approximate equilibrium is reached. Multilevel techniques are used to overcome local minima.

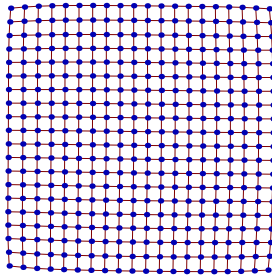
Spring embedding works particularly well for problems like regular grid graphs, in which it is possible to lay out the graph so that Euclidean distances between vertices are proportional to the graph distances.

This draws a 20×20 grid graph using the spring embedding algorithm.

```
In[11]:= << Combinatorica`;
```

```
In[12]:= GraphPlot[GridGraph[20, 20], Method -> "SpringEmbedding"]
```

```
Out[12]=
```



This method does, however, require more memory and CPU time. To reduce its $O(|V|^2)$ complexity, vertices that are far apart are ignored in the calculation of force and energy. See the method option "InferentialDistance" of GraphPlot and GraphPlot3D for more information.

Spring-Electrical Embedding

The disadvantage of the spring embedding algorithm is that it requires knowing the graph distance between every pair of vertices. Spring-electrical embedding uses two forces. The attractive force, $f_a = d_{ij}^2 / K$, is restricted to adjacent vertices and is proportional to the Euclidean distance between them, where K is related to the natural spring length. The electrical force, $f_r = -K^2 / d_{ij}$, on the other hand, is global and is inversely proportional to the Euclidean distance between nodes i and j . Overall, the energy to be minimized is $\sum_{i=1}^{|V|} f_i^2$, where

$$f_i = -C \sum_{j \neq i} \frac{K^2}{d_{ij}} \frac{(x_j - x_i)}{d_{ij}} + \sum_{i \leftrightarrow j} \frac{d_{ij}^2}{K} \frac{(x_j - x_i)}{d_{ij}} = -C \sum_{j \neq i} \frac{K^2}{d_{ij}^2} (x_j - x_i) + \sum_{i \leftrightarrow j} \frac{d_{ij}}{K} (x_j - x_i).$$

Here, C is a constant that regulates the relative strength of the repulsive and attractive forces, and $d_{ij} = \|x_i - x_j\|$ is the Euclidean distance between nodes i and j . For a graph of two vertices, the ideal Euclidean distance between the vertices is $K C^{1/3}$, which gives a total energy of zero.

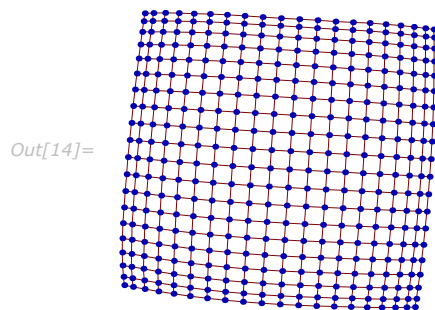
The layout of the graph vertices is calculated by minimizing the energy function. One way to do this is by iteratively moving each of the vertices along the direction of the spring force until an approximate equilibrium is reached. Multilevel techniques [7] are used to overcome local minima, and an octree data structure [16] is used to reduce the computational complexity in some cases.

In general, spring-electrical embedding works well for most problems. With multilevel and octree techniques, it is implemented very efficiently with a complexity of about $O(|V| \log(|V|))$.

This shows the drawing of a 20×20 grid graph using "SpringElectricalEmbedding".

```
In[13]:= << Combinatorica`;
```

```
In[14]:= GraphPlot[GridGraph[20, 20], Method -> "SpringElectricalEmbedding"]
```



A side effect of this algorithm is that vertices at the periphery tend to be closer to each other than those in the center, as seen in the previous drawing. This tendency can be alleviated with the method option "RepulsiveForcePower", which is described in "General Graph Drawing".

High-Dimensional Embedding Algorithm

In the high-dimensional embedding method, a graph is embedded in high-dimensional space, and then projected back to two- or three-dimensional space. First, a k -dimensional coordinate system is created based on k centers. The centers are a set of k vertices that are chosen to be as far apart as possible. The first vertex is selected at random, and then each of the remaining

centers is chosen as the farthest vertex from the previously selected centers. In other words, if j centers have been selected, c_{j+1} is the vertex whose shortest graph distance to the j centers is larger than or equal to the shortest graph distance of all the other vertices to the j centers.

With these k centers, a k -dimensional coordinate system can be established. Each vertex u_i has the coordinates $x_i = \{d_{u_i c_1}, d_{u_i c_2}, \dots, d_{u_i c_k}\}$, where $d_{u_i c_j}$ is the graph distance between the vertex u_i and the center c_j . The nk -dimensional coordinate vectors form an $n \times k$ matrix X , where x_i is the i^{th} row of X .

Since it is only possible to draw in two and three dimensions, and since the coordinates are correlated, the k -dimensional coordinates are projected back to two or three dimensions by a suitable linear combination. Assume that the graph with n coordinates and k centers is projected back to two dimensions. In order to make this projection shift-invariant, X is first normalized to X' .

$$X' = X - e e^T X / n, \quad e = \{1, \dots, 1\}$$

Let v_1 and v_2 be two k -dimensional vectors needed for the purpose of linear combination. The two linear combinations should be uncorrelated, so they must be orthogonal to each other.

$$(X' v_1)^T X' v_2 = v_1^T (X'^T X') v_2 = 0$$

Each must be as far away from 0 as possible.

$$v_i^T (X'^T X') v_i / \|v_i\|^2 \rightarrow \max, \quad i = 1, 2$$

To achieve this, you therefore select v_1 and v_2 to be the two eigenvectors that correspond to the first two largest eigenvalues of the $k \times k$ symmetric matrix $X'^T X'$. This process of choosing two highly uncorrelated vectors is also known as principal component analysis.

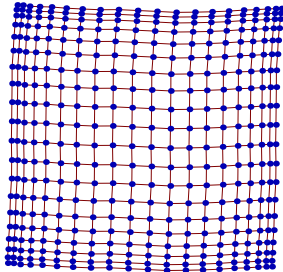
In summary, for two-dimensional drawing, the high-dimensional embedding method uses the coordinates of the vertices given by $X' v_1$ and $X' v_2$.

This shows the drawing of a 20×20 grid graph using "HighDimensionalEmbedding".

```
In[15]:= << Combinatorica`;
```

```
In[16]:= GraphPlot[GridGraph[20, 20], Method -> "HighDimensionalEmbedding"]
```

```
Out[16]=
```



The high-dimensional embedding method tends to be very fast but its results are often of lower quality than force-directed algorithms. The method can be specified with `Method -> "HighDimensionalEmbedding"` in `GraphPlot` and `GraphPlot3D`.

A Hierarchical Drawing Algorithm for Directed Graphs

The algorithm for drawing directed acyclic graphs (DAGs) follows the algorithm of Sugiyama et al. [14], and subsequent development [15]. It consists of the following stages:

1. Vertices of the DAG are first assigned a preliminary y ranking such that if there is an edge from i to j , then it is likely that $y\text{rank}(i) > y\text{rank}(j)$. This is to ensure that the final drawing has directed edges pointing mostly downward.
2. The y coordinates are generated so that if there is an edge from i to j and $y\text{rank}(i) > y\text{rank}(j)$, their y coordinates are as close as possible, but separated by a set minimum. This ensures that the final resulting drawing does not have many long edges. This process assigns the vertices into a finite number of layers. If an edge lies across a number of layers, virtual vertices are added.
3. A preliminary x ranking is assigned to each vertex to minimize the number of edge crossings.
4. The x coordinates are generated by minimizing $\sum_{i \rightarrow j} |x(i) - x(j)|$ subject to the constraints that vertices on the same layer obey the x ranking generated in step 3 and are separated by a set minimum.

The resulting drawing lays out the graph in a hierarchical structure, where most of the edges point downward. `LayeredGraphPlot` function implements this algorithm.

Algorithms for Drawing Trees

Two algorithms for drawing trees are the radial drawing algorithm and the layered drawing algorithm [1]. In the radial drawing algorithm, a reasonable root of the tree is chosen. Then, starting from that root of the tree, each subtree is drawn inside a wedge, with the angle of the wedge proportional to the number of leaves in that subtree. In the layered drawing algorithm, a reasonable root of the tree is chosen. Then, starting from that root, subtrees of the root are recursively drawn such that vertices on the same level have the same y coordinate, and the horizontally closest vertices of adjacent subtrees are of unit distance apart. The root is placed at the center of the x coordinates of its subtrees and its y coordinate is one unit above them. `TreePlot` function chooses between these two algorithms, depending on the second argument of this function.

Selecting the Appropriate Graph Drawing Function

For general graph drawing, consider using `GraphPlot` or `GraphPlot3D`. `GraphPlot` or `GraphPlot3D` calculates a visually appealing 2D/3D layout and plots the graph using this layout. See "General Graph Drawing" for these functions, and [17] for algorithmic details.

To get a layered/hierarchical drawing of a directed graph, use `LayeredGraphPlot`. `LayeredGraphPlot` attempts to draw the vertices of a graph in a series of layers, with dominant vertices at the top, and vertices lower in the hierarchy progressively farther down. This function is most suitable for applications such as flow chart drawing. See "Hierarchical Drawing of Directed Graphs" for this function.

`TreePlot` is specifically designed to draw trees and tree-like graphs. See "Tree Drawing" for this function.

References

- [1] Di Battista, G., P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.
- [2] Fruchterman, T. M. J. and E. M. Reingold. "Graph Drawing by Force-Directed Placement." *Software—Practice and Experience* 21, no. 11 (1991): 1129-1164.
- [3] Eades, P. "A Heuristic for Graph Drawing." *Congressus Numerantium* 42 (1984): 149-160.
- [4] Quinn, N. and M. Breuer. "A Force Directed Component Placement Procedure for Printed Circuit Boards." *IEEE Trans. on Circuits and Systems* 26, no. 6 (1979): 377-388.
- [5] Kamada, T. and S. Kawai. "An Algorithm for Drawing General Undirected Graphs." *Information Processing Letters* 31 (1989): 7-15.
- [6] Harel, D. and Y. Koren. "Graph Drawing by High-Dimensional Embedding." In *Proceedings of 10th Int. Symp. Graph Drawing (GD'02)*, 207-219, 2002.
- [7] Walshaw, C. "A Multilevel Algorithm for Force-Directed Graph-Drawing." *J. Graph Algorithms Appl.* 7, no. 3 (2003): 253-285.
- [8] Cuthill, E. and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices." In *Proceedings, 24th National Conference of ACM*, 157-172, 1969.
- [9] Lim, A., B. Rodrigues, and F. Xiao. "A Centroid-Based Approach to Solve the Bandwidth Minimization Problem." In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, 30075.1, 2004.
- [10] Barnard, S. T., A. Pothen, and H. D. Simon. "A Spectral Algorithm for Envelope Reduction of Sparse Matrices." *Journal of Numerical Linear Algebra with Applications* 2, no. 4 (1995): 317-334.
- [11] Sloan, S. "A Fortran Program for Profile and Wavefront Reduction." *International Journal for Numerical Methods in Engineering* 28, no. 11 (1989): 2651-2679.
- [12] Reid, J. K. and J. A. Scott. "Ordering Symmetric Sparse Matrices for Small Profile and Wavefront." *International Journal for Numerical Methods in Engineering* 45, no. 12 (1999): 1737-1755.

- [13] George, J. A. "Computer Implementation of the Finite-Element Method." Report STAN CS-71-208, PhD Thesis, Department of Computer Science, Stanford University, Stanford, California, 1971.
- [14] Sugiyama, K., S. Tagawa, and M. Toda. "Methods for Visual Understanding of Hierarchical Systems." *IEEE Trans. Syst. Man, Cybern.* 11, no. 2 (1981): 109-125.
- [15] Gansner, E. R., E. Koutsofios, S. C. North, and K. P. Vo. "A Technique for Drawing Directed Graphs." *IEEE Trans. Software Engineering* 19, no. 3 (1993): 214-230.
- [16] Quigley, A. "Large Scale Relational Information Visualization, Clustering, and Abstraction." PhD Thesis, Department of Computer Science and Software Engineering, University of Newcastle, Australia, 2001.
- [17] Hu, Y. F. "Efficient, High-Quality Force-Directed Graph Drawing." *The Mathematica Journal* 10, no. 1 (2006): 37-71.

General Graph Drawing

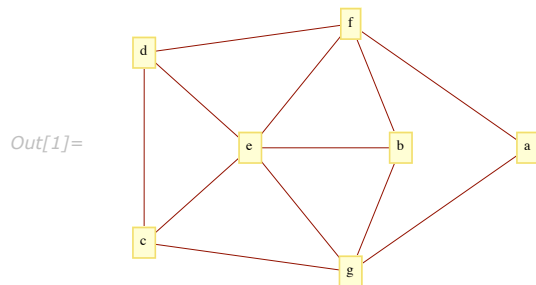
`GraphPlot` and `GraphPlot3D` calculate and plot a visually appealing 2D/3D layout of a graph. The functions are designed to work with very large graphs and handle both connected and disconnected graphs.

<code>GraphPlot</code> [{ { $v_{i1} \rightarrow v_{j1}$, $v_{i2} \rightarrow v_{j2}$, ... } }]	generate a plot of the graph in which vertex v_{ik} is connected to vertex v_{jk}
<code>GraphPlot</code> [{ { $v_{i1} \rightarrow v_{j1}$, lbl_1 } , ... }]	associate labels lbl_k with edges in the graph
<code>GraphPlot</code> [m]	generate a plot of the graph represented by the adjacency matrix m
<code>GraphPlot3D</code> [{ { $v_{i1} \rightarrow v_{j1}$, $v_{i2} \rightarrow v_{j2}$, ... } }]	generate a 3D plot of the graph in which vertex v_{ik} is connected to vertex v_{jk}
<code>GraphPlot3D</code> [{ { $v_{i1} \rightarrow v_{j1}$, lbl_1 } , ... }]	associate labels lbl_k with edges in the graph
<code>GraphPlot3D</code> [m]	generate a 3D plot of the graph represented by the adjacency matrix m

Graph drawing functions.

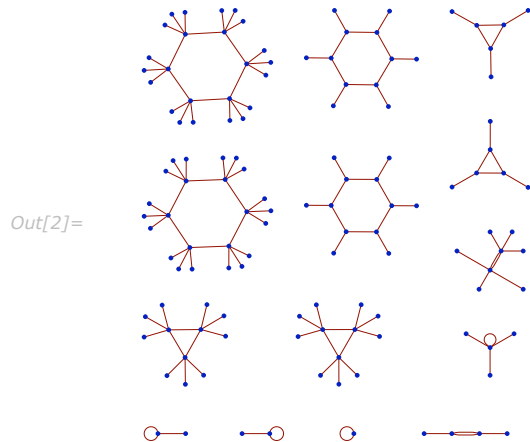
This plots a graph specified by a rule list.

```
In[1]:= GraphPlot[{"d" -> "c", "e" -> "b", "e" -> "c", "e" -> "d", "f" -> "a", "f" -> "b", "f" -> "d",  
"f" -> "e", "g" -> "a", "g" -> "b", "g" -> "c", "g" -> "e"}, VertexLabeling -> True]
```



For disconnected graphs, individual components are laid out in a visually appealing way and assembled.

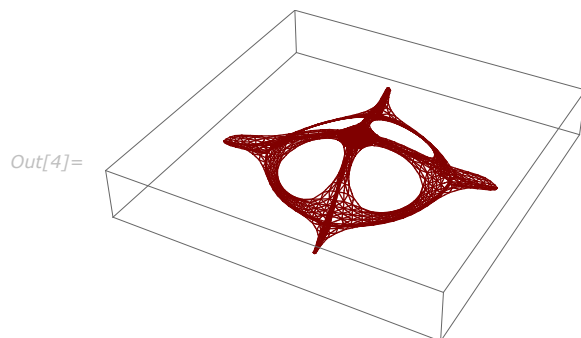
```
In[2]:= GraphPlot[Table[i → Mod[i^2, 129], {i, 0, 128}]]
```



This is a larger graph defined by a sparse adjacency matrix from a structural engineering application. The matrix comes from the Harwell-Boeing Collection.

```
In[3]:= a = Import["LinearAlgebraExamples/Data/dwt_1005.psa", "HarwellBoeing"];
```

```
In[4]:= GraphPlot3D[a, VertexRenderingFunction → None]
```



GraphPlot may produce slightly different output on different platforms, due to floating-point differences.

Options for GraphPlot and GraphPlot3D

The following options are accepted for GraphPlot and GraphPlot3D (DirectedEdges and EdgeLabeling options are only valid for GraphPlot); in addition, options for Graphics and Graphics3D are accepted.

<i>option name</i>	<i>default value</i>	
DirectedEdges	True	whether to show edges as directed arrows
EdgeLabeling	True	whether to include labels given for edges
EdgeRenderingFunction	Automatic	function to give explicit graphics for edges
Method	Automatic	the method used to lay out the graph
MultiedgeStyle	Automatic	how to draw multiple edges between vertices
PlotRangePadding	Automatic	how much padding to put around the plot
PackingMethod	Automatic	method to use for packing components
PlotStyle	Automatic	style in which objects are drawn
SelfLoopStyle	Automatic	how to draw edges linking a vertex to itself
VertexCoordinateRules	Automatic	rules for explicit vertex coordinates
VertexLabeling	Automatic	whether to show vertex names as labels
VertexRenderingFunction	Automatic	function to give explicit graphics for vertices

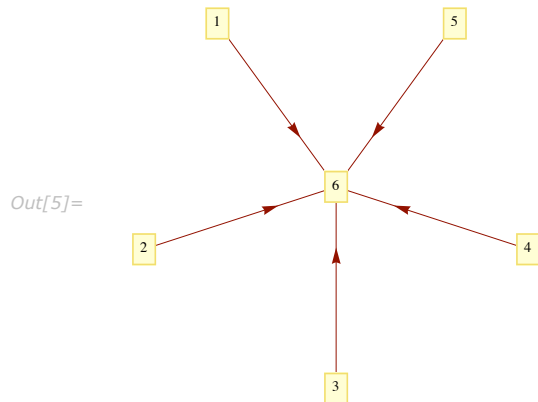
Options for GraphPlot and GraphPlot3D.

DirectedEdges

The option `DirectedEdges` specifies whether to draw edges as arrows. Possible values for this option are `True` or `False`. The default value for this option is `False`.

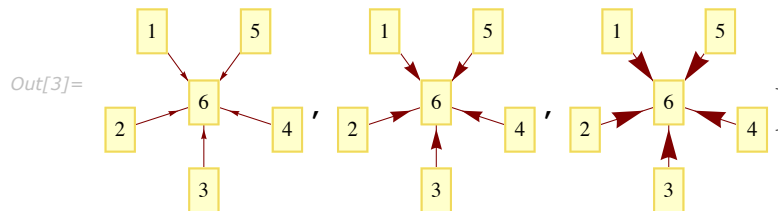
This shows a graph with edges represented by arrows.

```
In[5]:= GraphPlot[{1 → 6, 2 → 6, 3 → 6, 4 → 6, 5 → 6},
  DirectedEdges → True, VertexLabeling → True]
```



This makes the arrowheads larger.

```
In[3]:= Table[
  GraphPlot[{1 → 6, 2 → 6, 3 → 6, 4 → 6, 5 → 6},
    ImageSize → 100, DirectedEdges → {True, "ArrowheadsSize" → asize},
    VertexLabeling → True], {asize, {0.05, 0.1, 0.15}}
```

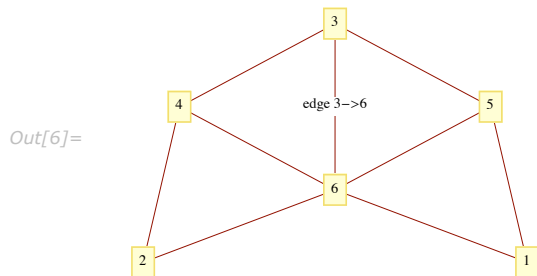


EdgeLabeling

The option `EdgeLabeling` specifies whether and how to display labels given for the edges. Possible values for this option are `True`, `False`, or `Automatic`. The default value for this option is `True`, which displays the supplied edge labels on the graph. With `EdgeLabeling -> Automatic`, the labels are shown as tooltips.

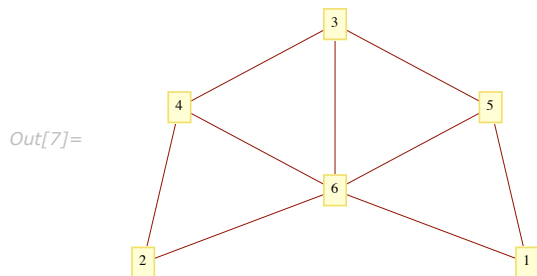
By default, `GraphPlot` displays the supplied label for the edge between vertices 3 and 6.

```
In[6]:= GraphPlot[{1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4, 3 → 5, {3 → 6, "edge 3->6"}, 4 → 6, 5 → 6},
  VertexLabeling → True]
```



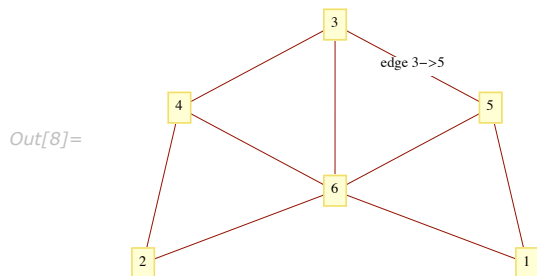
This displays the edge label as a tooltip. Place the cursor over the edge between vertices 3 and 6 to see the tooltip.

```
In[7]:= GraphPlot[{1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4, 3 → 5, {3 → 6, "edge 3->6"}, 4 → 6, 5 → 6},
  EdgeLabeling → Automatic, VertexLabeling → True]
```



Alternatively, use `Tooltip[vi -> vj, lbl]` to specify a tooltip for an edge. Place the cursor over the edge between vertices 3 and 6, as well as over the edge label on the edge between vertices 3 and 5, to see the tooltips.

```
In[8]:= GraphPlot[{1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4, {3 → 5, Tooltip["edge 3->5", "3->5"]},
  Tooltip[3 → 6, "3->6"], 4 → 6, 5 → 6}, VertexLabeling → True]
```



To display the supplied label for the edge in 3D, `EdgeRenderingFunction` needs to be used. This is described in "Edge Rendering Function".

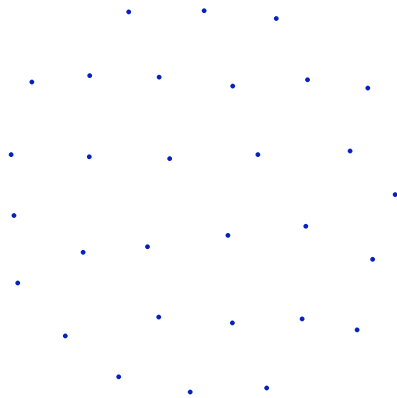
EdgeRenderingFunction

The option `EdgeRenderingFunction` specifies graphical representation of the graph edges. Possible values for this option are `Automatic`, `None`, or a function that gives a proper combination of graphics primitives and directives. With the default setting of `Automatic`, a dark red line is drawn for each edge. With `EdgeRenderingFunction -> None`, edges are not drawn.

This draws vertices only.

```
In[9]:= GraphPlot[Table[1, {30}, {30}], EdgeRenderingFunction -> None]
```

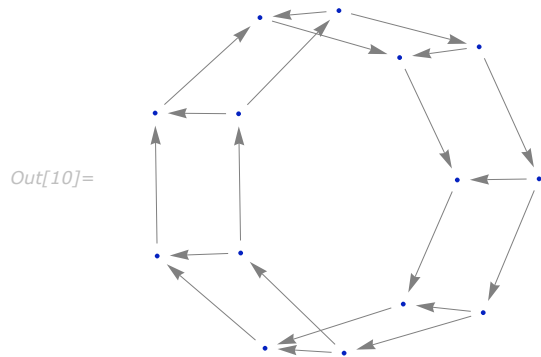
Out[9]=



With `EdgeRenderingFunction -> g`, each edge is rendered with the graphics primitives and directives given by the function g that can take three or more arguments, in the form $g[\{r_i, \dots, r_j\}, \{v_i, v_j\}, lbl_{ij}, \dots]$, where r_i, r_j are the coordinates of the beginning and ending points of the edge, v_i, v_j are the beginning and ending vertices, and lbl_{ij} is any label specified for the edge or `None`. Explicit settings for `EdgeRenderingFunction -> g` override settings for `EdgeLabeling` and `DirectedEdges`.

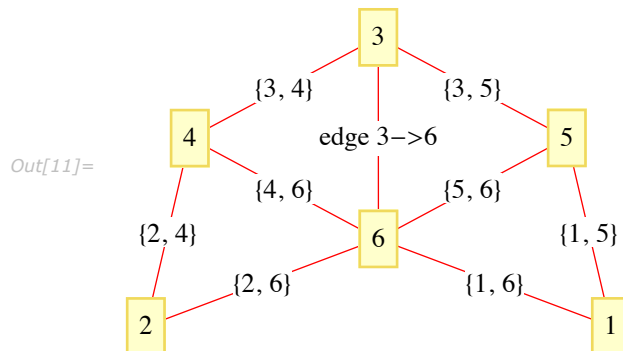
This plots the edges as gray arrows with ends set back from vertices by a distance 0.1 (in the graph's coordinate system).

```
In[10]:= GraphPlot[{1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 7,
  7 → 1, 11 → 12, 12 → 13, 13 → 14, 14 → 15, 15 → 16, 16 → 17,
  17 → 11, 1 → 11, 2 → 12, 3 → 13, 4 → 14, 5 → 15, 6 → 16, 7 → 17},
  EdgeRenderingFunction → ({GrayLevel[0.5], Arrow[#1, 0.1]} &)]
```



This generates edge labels or displays the ones supplied in the description of the graph.

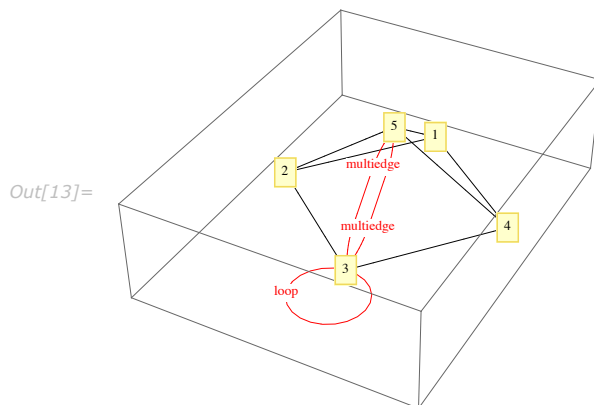
```
In[11]:= GraphPlot[{1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4, 3 → 5, {3 → 6, "edge 3->6"}, 4 → 6, 5 → 6},
  EdgeRenderingFunction → ({Red, Line[#1]}, Text[If[#3 === None, #2, #3],
  Total[#1] / 2., Background → White]} &), VertexLabeling → True]
```



This draws straight edges in black and other edges (two multiedges and a self-loop) in red. The function `LineScaledCoordinate` from the package `GraphUtilities` is used to place labels at 70% of the length of the edge.

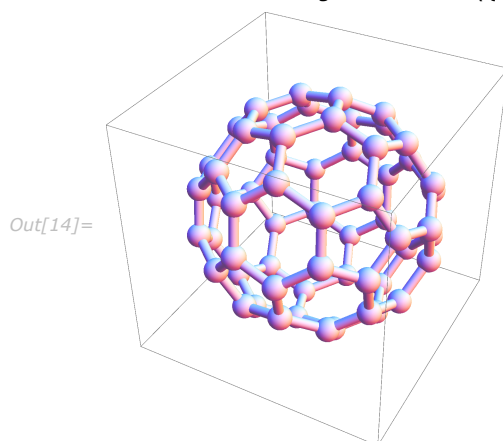
```
In[12]:= << GraphUtilities`
```

```
In[13]:= GraphPlot3D[{1 → 2, 2 → 3, 3 → 4, 5 → 1, 5 → 2, 5 → 3, 5 → 4, 1 → 4, 3 → 5, 3 → 3},
  EdgeRenderingFunction →
    (If [Length[#1] > 2, {Red, Line[#1], Text[If[First[#1] === Last[#1],
      "loop", "multiedge"], LineScaledCoordinate[#1, .7],
      Background → White]}, Line[#1]] &), VertexLabeling → True]
```



This plots a 3D graph using spheres for vertices and cylinders for edges.

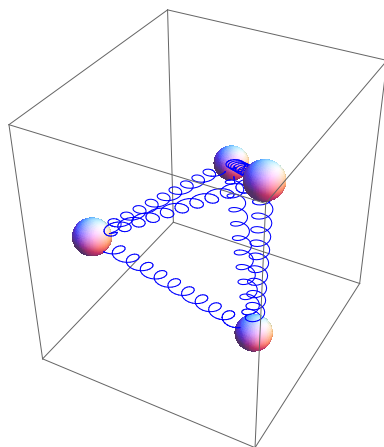
```
In[14]:= GraphPlot3D[{2 → 1, 5 → 1, 6 → 1, 3 → 2, 11 → 2, 4 → 3, 16 → 3, 5 → 4, 21 → 4,
  26 → 5, 7 → 6, 10 → 6, 8 → 7, 30 → 7, 9 → 8, 42 → 8, 10 → 9, 38 → 9, 12 → 10,
  12 → 11, 15 → 11, 13 → 12, 14 → 13, 37 → 13, 15 → 14, 33 → 14, 17 → 15, 17 → 16,
  20 → 16, 18 → 17, 19 → 18, 32 → 18, 20 → 19, 53 → 19, 22 → 20, 22 → 21, 25 → 21,
  23 → 22, 24 → 23, 52 → 23, 25 → 24, 48 → 24, 27 → 25, 27 → 26, 30 → 26, 28 → 27,
  29 → 28, 47 → 28, 30 → 29, 43 → 29, 32 → 31, 35 → 31, 54 → 31, 33 → 32, 34 → 33,
  35 → 34, 36 → 34, 56 → 35, 37 → 36, 40 → 36, 38 → 37, 39 → 38, 40 → 39, 41 → 39,
  57 → 40, 42 → 41, 45 → 41, 43 → 42, 44 → 43, 45 → 44, 46 → 44, 58 → 45, 47 → 46,
  50 → 46, 48 → 47, 49 → 48, 50 → 49, 51 → 49, 59 → 50, 52 → 51, 55 → 51, 53 → 52,
  54 → 53, 55 → 54, 60 → 55, 57 → 56, 60 → 56, 58 → 57, 59 → 58, 60 → 59},
  EdgeRenderingFunction → ({Cylinder[#1, .1]} &),
  VertexRenderingFunction → ({Sphere[#, .25]} &)]
```



This plots a graph with edges displayed as springs and vertices as spheres.

```
In[15]:= Spring[{x0_, y0_}, n_: 10] :=
Module[{x = x0 + (y0 - x0) * 0.05, y = y0 + (x0 - y0) * 0.1, theta, t},
theta = If[(y0 - x0)[[1]] == 0., Pi / 2., ArcTan[(y0 - x0)[[2]] / ((y0 - x0)[[1]])]];
Line[Join[{x0}, Table[x + (y - x) * t + 0.05
{Cos[2 Pi n t + theta], Sin[2 Pi n t + theta]}, 0}, {t, 0, 1, .005}], {y0}]]];
GraphPlot3D[{1 -> 2, 2 -> 3, 3 -> 1, 1 -> 4, 2 -> 4, 3 -> 4}, EdgeRenderingFunction ->
({Blue, Spring[#1]} &), VertexRenderingFunction -> (Sphere[#1, 0.1] &)]
```

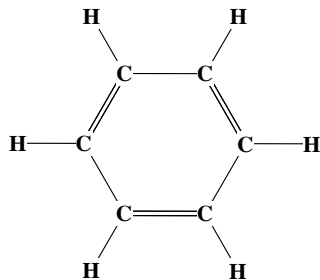
Out[16]=



This plots the benzene molecule.

```
In[17]:= normal[{x_, y_}] := 0.03 * {-y, x} / Norm[{x, y}];
GraphPlot[{1 -> 2, 2 -> 3, 3 -> 4, 4 -> 5, 5 -> 6, 6 -> 1, 1 -> 2, 3 -> 4, 5 -> 6,
1 -> 7, 2 -> 8, 3 -> 9, 4 -> 10, 5 -> 11, 6 -> 12}, VertexRenderingFunction ->
(Text[Style[If[#2 <= 6, "C", "H"], Bold], #1, Background -> White] &),
EdgeRenderingFunction -> (If[Length[#1] > 2, norm = normal[First[#1] - Last[#1]];
{Line[{First[#1] + norm, Last[#1] + norm}],
Line[{First[#1] - norm, Last[#1] - norm}]}, Line[#1]] &)]
```

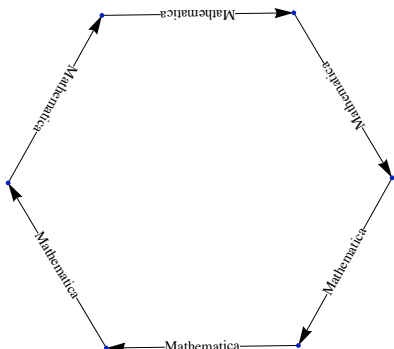
Out[18]=



Draw text along the edges.

```
In[11]:= GraphPlot[{1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 1},
  EdgeRenderingFunction → ({Arrow[#], Inset["Mathematica", Mean[#1],
  Automatic, Automatic, #[[1]] - #[[2]], Background → White]} &)]
```

Out[11]=

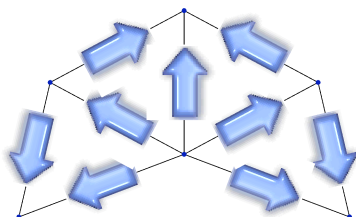


Draw graphics on each edge.

```
In[5]:= arrow =  ;
```

```
In[10]:= GraphPlot[{1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4, 3 → 5, 3 → 6, 4 → 6, 5 → 6},
  EdgeRenderingFunction → ({Line[#], Inset[arrow, Mean[#1],
  Automatic, Automatic, #[[1]] - #[[2]], Background → White]} &)]
```

Out[10]=



Method

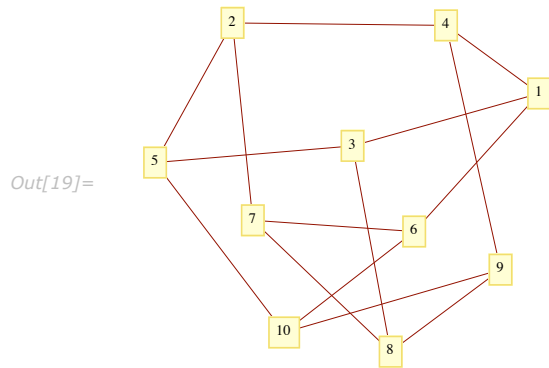
Algorithms to be used in `GraphPlot` and `GraphPlot3D` can be specified using the `Method` option, either as `Method -> "name"` or `Method -> {"name", opt1 -> val1, ...}`, where `opti -> vali` are method-specific options, described in separate sections. `Method -> Automatic` uses the "RadialDrawing" method for trees and "SpringElectricalEmbedding" otherwise.

<code>Automatic</code>	a method suitable for the problem is chosen automatically
<code>"CircularEmbedding"</code>	lay out the vertices in a circle
<code>"HighDimensionalEmbedding"</code>	invoke the high-dimensional embedding method, in which the graph is first laid out in a high-dimensional space based on the graph distances of the vertices to k centers; this layout is then projected to 2D or 3D space by linear combination of the high-dimensional coordinates using principal component analysis
<code>"RadialDrawing"</code>	invoke the radial drawing method, which is most suitable for tree or tree-like graphs; if the graph is not a tree, a spanning tree is first constructed, and a radial drawing of the spanning tree is used to derive the drawing for the graph
<code>"RandomEmbedding"</code>	lay out vertices randomly
<code>"SpiralEmbedding"</code>	lay out the vertices in a spiral; in 3D, this distributes vertices uniformly on a sphere
<code>"SpringElectricalEmbedding"</code>	invoke the spring-electrical embedding method, in which neighboring vertices are subject to an attractive spring force that is proportional to their physical distance, and all vertices are subject to a repulsive electrical force that is inversely proportional to their distance; the overall energy is minimized
<code>"SpringEmbedding"</code>	invoke the spring embedding method, in which a vertex is subject to either attractive or repulsive force from another vertex, as though they are connected by a spring; the spring has an ideal length equal to the graph distance between the vertices; the total spring energy is minimized

Valid values of the `Method` option.

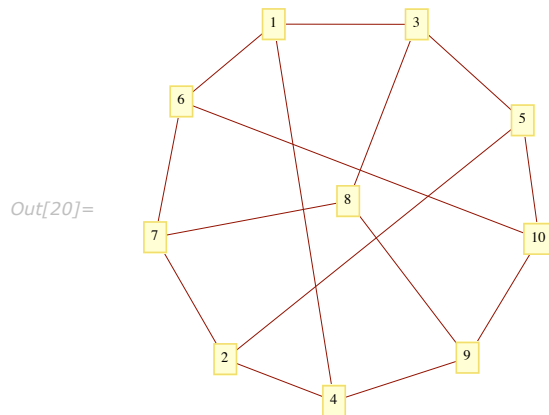
This draws the Petersen graph using the default method.

```
In[19]:= GraphPlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7, 7 → 8, 8 → 9,
  9 → 10, 6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10}, VertexLabeling → True]
```



This draws the graph using the "SpringEmbedding" algorithm.

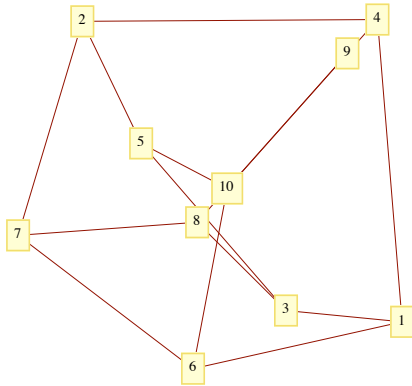
```
In[20]:= GraphPlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7, 7 → 8, 8 → 9, 9 → 10, 6 → 10, 1 → 6,
  2 → 7, 3 → 8, 4 → 9, 5 → 10}, Method → "SpringEmbedding", VertexLabeling → True]
```



This draws the graph using the "HighDimensionalEmbedding" algorithm.

```
In[21]:= GraphPlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7,
  7 → 8, 8 → 9, 9 → 10, 6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10},
  Method → "HighDimensionalEmbedding", VertexLabeling → True]
```

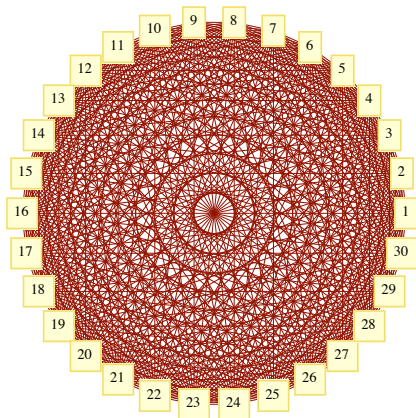
Out[21]=



This draws the complete graph of 30 vertices using the "CircularEmbedding" method.

```
In[22]:= GraphPlot[Table[1, {30}], {30}],
  Method → "CircularEmbedding", VertexLabeling → True]
```

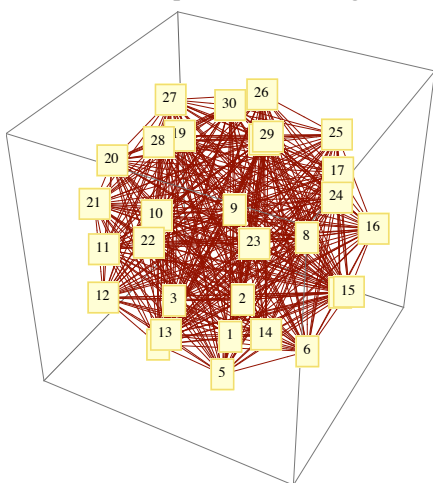
Out[22]=



This draws the complete graph of 30 vertices using the "SpiralEmbedding" method in 3D.

```
In[23]:= GraphPlot3D[Table[1, {30}, {30}],  
Method → "SpiralEmbedding", VertexLabeling → True]
```

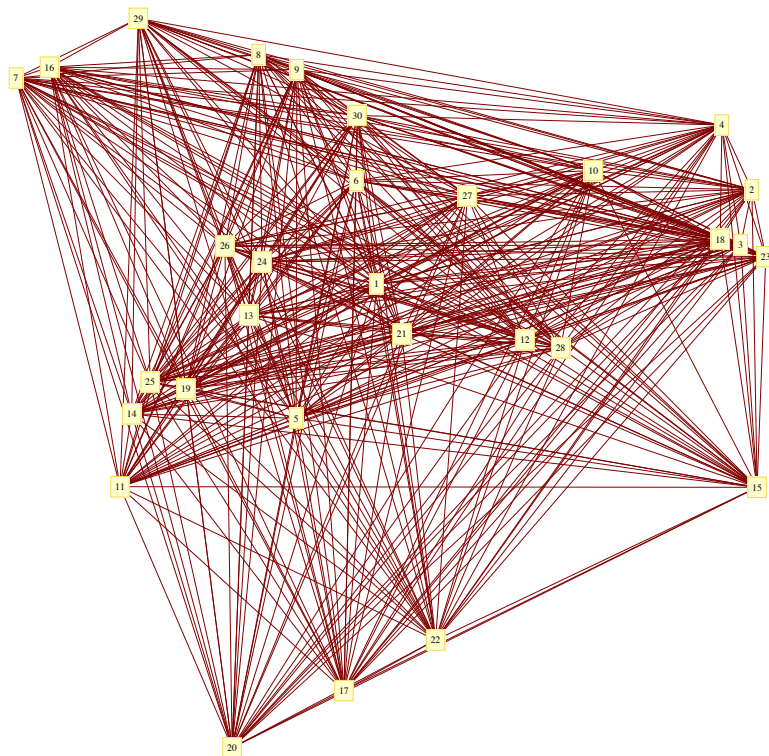
Out[23]=



This draws the complete graph using the "RandomEmbedding" method.

```
In[24]:= GraphPlot[Table[1, {30}, {30}], Method → "RandomEmbedding", VertexLabeling → True]
```

Out[24]=

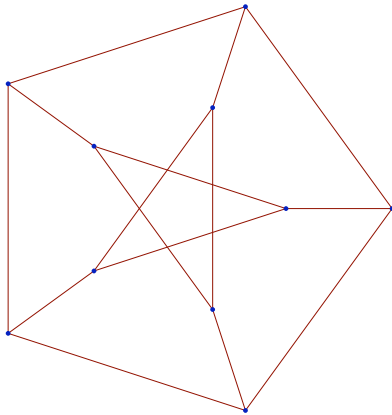


This draws a graph specified in the *Combinatorica* Package format using coordinates that come with it.

```
In[25]:= Needs["Combinatorica`"]
```

```
In[26]:= GraphPlot[PetersenGraph, Method -> None]
```

```
Out[26]=
```

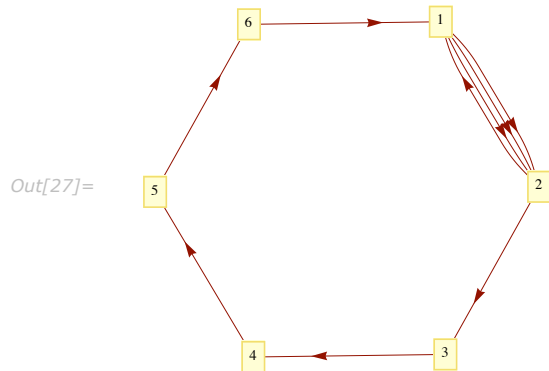


MultiedgeStyle

The option `MultiedgeStyle` specifies whether to draw multiple edges between two vertices. Possible values for `MultiedgeStyle` are `Automatic` (the default), `True`, `False`, or a positive real number. With the default setting `MultiedgeStyle -> Automatic`, multiple edges are shown for a graph specified by a list of rules, but not shown if specified by an adjacency matrix. With `MultiedgeStyle -> δ` , the multiedges are spread out to a scaled distance of δ .

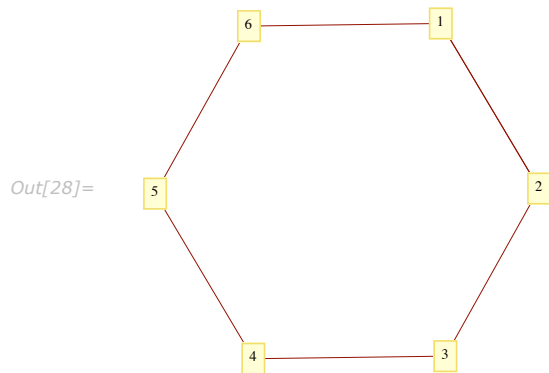
By default, multiple edges are shown if a graph is given as a list of rules.

```
In[27]:= GraphPlot[{1 → 2, 2 → 1, 1 → 2, 1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 1},
  DirectedEdges → True, VertexLabeling → True]
```



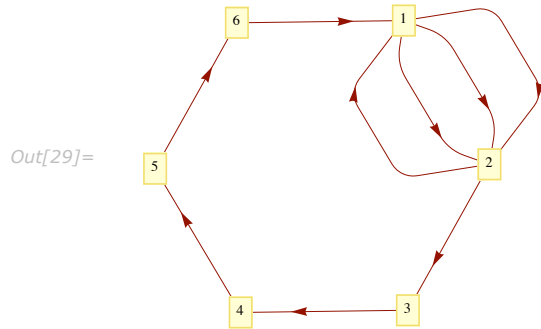
But multiple edges are not shown for graphs specified by an adjacency matrix.

```
In[28]:= GraphPlot[ $\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ , VertexLabeling → True]
```



This spreads multiple edges by the specified amount.

```
In[29]:= GraphPlot[{1 → 2, 2 → 1, 1 → 2, 1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 1},  
  MultiedgeStyle → 1, DirectedEdges → True, VertexLabeling → True]
```

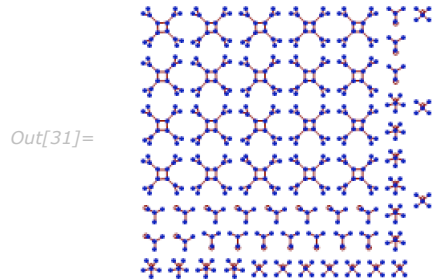


PackingMethod

The option `PackingMethod` specifies the method used for packing disconnected components. Possible values for the option are `Automatic` (the default), `"ClosestPacking"`, `"ClosestPackingCenter"`, `"Layered"`, `"LayeredLeft"`, `"LayeredTop"`, and `"NestedGrid"`. With `PackingMethod -> "ClosestPacking"`, components are packed as close together as possible using a polyomino method [6], starting from the top left. With `PackingMethod -> "ClosestPackingCenter"`, components are packed starting from the center. With `PackingMethod -> "Layered"`, components are packed in layers starting from top left. With `PackingMethod -> "LayeredLeft"` or `PackingMethod -> "LayeredTop"`, components are packed in layers starting from the top/left respectively. With `PackingMethod -> "NestedGrid"`, components are arranged in a nested grid. The typical effective default setting is `PackingMethod -> "Layered"`, and the packing starts with the components of the largest bounding box area.

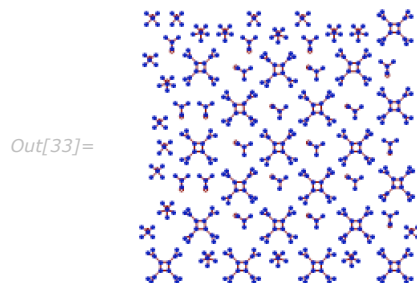
This shows the packing of disconnected components by the default method.

```
In[30]:= g = Flatten[Table[x = RandomReal[1, {50 + 1}];
  Table[x[[i + 1]] -> x[[Mod[i^2, 50] + 1]], {i, 0, 50}], {10}];
In[31]:= GraphPlot[g]
```



This shows the packing of disconnected components using the "ClosestPackingCenter" method.

```
In[32]:= g = Flatten[Table[x = RandomReal[1, {50 + 1}];
  Table[x[[i + 1]] -> x[[Mod[i^2, 50] + 1]], {i, 0, 50}], {10}];
In[33]:= GraphPlot[g, PackingMethod -> "ClosestPackingCenter"]
```



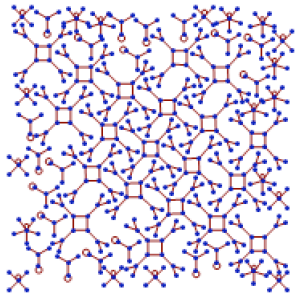
Users can adjust the packing by suboptions of `PackingMethod`. The suboption `"Padding"` specifies the amount of space to allow between components; possible values are `Automatic` (the default), or a non-negative number. The suboption `"PaddingFunction"`, which overrides `"Padding"`, also specifies the amount of space to allow between components. It takes a list of the form $\{\{w_1, h_1\}, \dots\}$, which are the width and height of the bounding box of the components, and returns a non-negative number. Options `PackingMethod -> "ClosestPacking"` and `PackingMethod -> "ClosestPackingCenter"` also accept a `"PolyominoNumber"` suboption, which specifies the average number of polyominoes used to approximate each disconnected

component. Possible values for the "PolyominoNumber" suboption are Automatic (the default, which usually sets "PolyominoNumber" to 100), or a positive integer. A smaller "PolyominoNumber" typically has the effect of not allowing smaller components to embed in between large components.

This specifies a space of one polyomino between components.

```
In[34]:= g = Flatten[Table[x = RandomReal[1, {50 + 1}];
  Table[x[[i + 1]] -> x[[Mod[i^2, 50] + 1]], {i, 0, 50}], {10}];
In[35]:= GraphPlot[g, PackingMethod -> {"ClosestPackingCenter", "Padding" -> 1}]
```

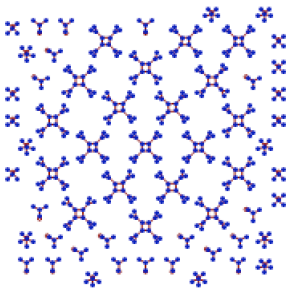
Out[35]=



This specifies that an average of five polyominos be used to approximate each component.

```
In[36]:= g = Flatten[Table[x = RandomReal[1, {50 + 1}];
  Table[x[[i + 1]] -> x[[Mod[i^2, 50] + 1]], {i, 0, 50}], {10}];
In[37]:= GraphPlot[g,
  PackingMethod -> {"ClosestPackingCenter", "PolyominoNumber" -> 5, "Padding" -> 1}]
```

Out[37]=



PlotRangePadding

`PlotRangePadding` is a common option for graphics functions inherited by `GraphPlot` and `GraphPlot3D`.

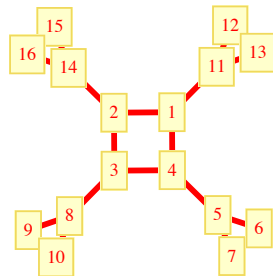
PlotStyle

`PlotStyle` is a common option for graphics functions inherited by `GraphPlot` and `GraphPlot3D`. The option `PlotStyle` specifies the style in which objects are drawn.

Draw edges with thicker lines, and both edges and vertex labels in red.

```
In[38]:= GraphPlot[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 6 → 5, 7 → 5, 5 → 4, 9 → 8,
  10 → 8, 8 → 3, 12 → 11, 13 → 11, 11 → 1, 15 → 14, 16 → 14, 14 → 2},
  VertexLabeling → True, PlotStyle → {Red, Thickness[0.02]}]
```

Out[38]=

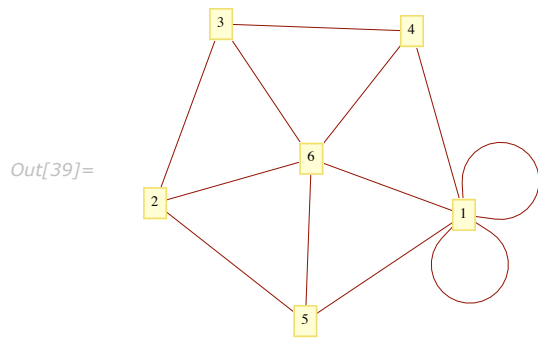


SelfLoopStyle

The option `SelfLoopStyle` specifies whether and how to draw loops for vertices that are linked to themselves. Possible values of the option are `Automatic` (the default), `True`, `False`, or a positive real number. With `SelfLoopStyle -> Automatic`, self-loops are shown if the graph is specified by a list of rules, but not by an adjacency matrix. With `SelfLoopStyle -> δ` , the self-loops are drawn with a diameter of δ (relative to the average edge length).

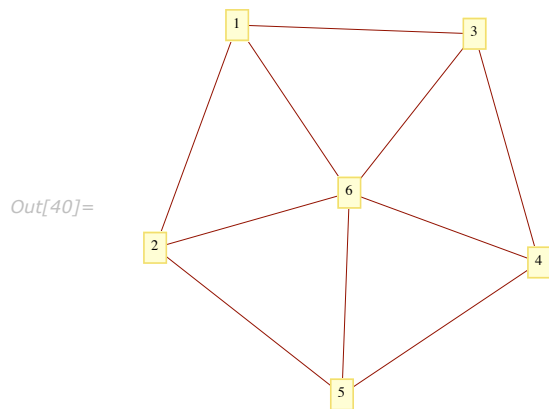
By default, self-loops are displayed for a graph specified by a list of rules.

```
In[39]:= GraphPlot[{3 → 2, 4 → 1, 4 → 3, 5 → 1, 5 → 2, 6 → 1,
  6 → 2, 6 → 3, 6 → 4, 6 → 5, 1 → 1, 1 → 1}, VertexLabeling → True]
```



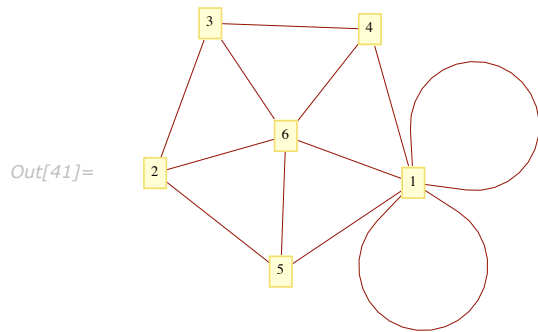
Self-loops are not shown if the graph is specified by an adjacency matrix.

```
In[40]:= GraphPlot  $\left[ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \right]$ , VertexLabeling → True]
```



This shows self-loops with the diameter as big as the average length of the edges.

```
In[41]:= GraphPlot[{3 → 2, 4 → 1, 4 → 3, 5 → 1, 5 → 2, 6 → 1, 6 → 2, 6 → 3,
6 → 4, 6 → 5, 1 → 1, 1 → 1}, VertexLabeling → True, SelfLoopStyle → 1]
```

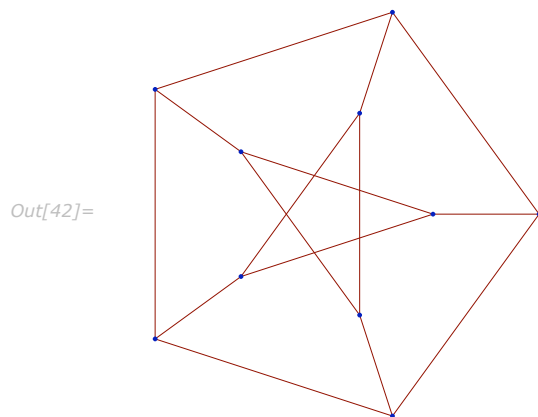


VertexCoordinateRules

The option `VertexCoordinateRules` specifies the coordinates of the vertices. Possible values are `None`, a list of coordinates, or a list of rules specifying the coordinates of selected or all vertices.

This draws the Petersen graph using known coordinates.

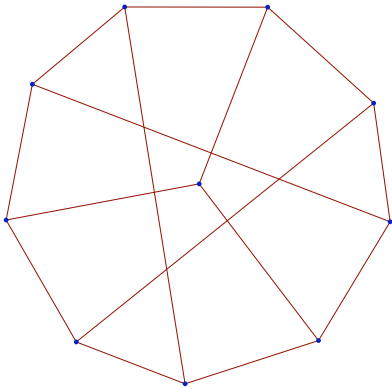
```
In[42]:= GraphPlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7, 7 → 8, 8 → 9, 9 → 10,
6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10}, VertexCoordinateRules →
{{0.309, 0.951}, {-0.809, -0.587}, {0.309, -0.951}, {-0.809, 0.587}, {1., 0},
{0.618, 1.902}, {-1.618, 1.175}, {-1.618, -1.175}, {0.618, -1.902}, {2., 0}}]
```



This computes vertex coordinates of the same graph using the "SpringEmbedding" algorithm.

```
In[43]:= GraphPlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7, 7 → 8, 8 → 9, 9 → 10,
  6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10}, Method → "SpringEmbedding"]
```

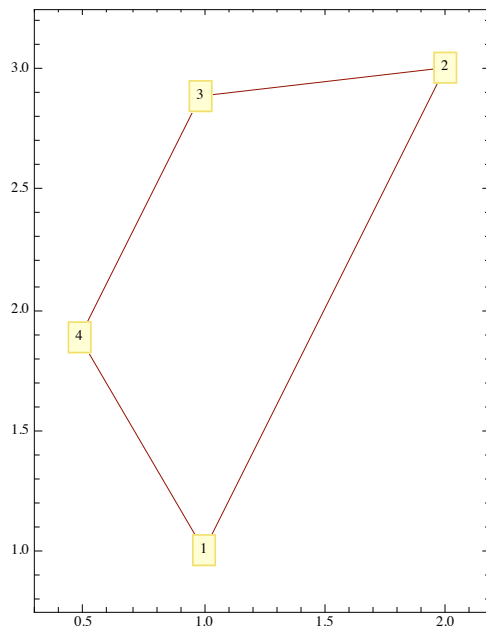
Out[43]=



This specifies coordinates for two vertices.

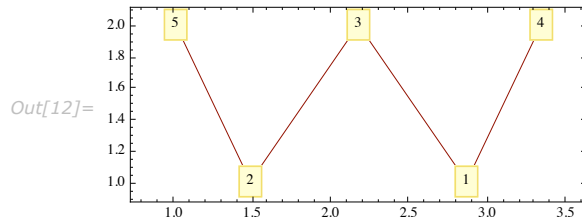
```
In[44]:= GraphPlot[{1 → 2, 2 → 3, 3 → 4, 4 → 1},
  VertexCoordinateRules → {1 → {1, 1}, 2 → {2, 3}},
  Frame → True, FrameTicks → True, VertexLabeling → True]
```

Out[44]=



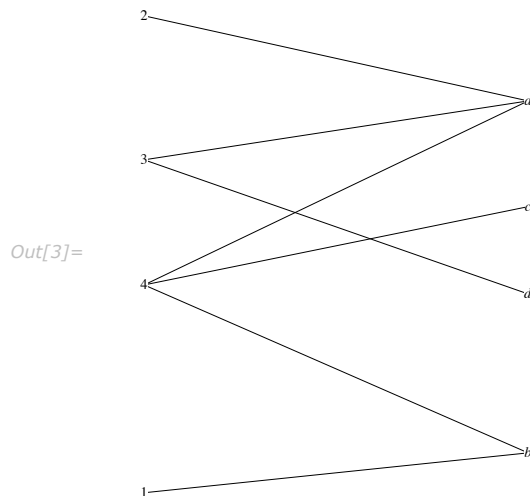
This specifies only y coordinates.

```
In[12]:= GraphPlot[{1 → 4, 2 → 3, 2 → 5, 1 → 3},
  VertexCoordinateRules → {1 → {Automatic, 1}, 2 → {Automatic, 1},
    3 → {Automatic, 2}, 4 → {Automatic, 2}, 5 → {Automatic, 2}},
  Frame → True, FrameTicks → True, VertexLabeling → True]
```



This draws a bipartite graph by fixing x coordinates. "Anchors" are added to connect disconnected components.

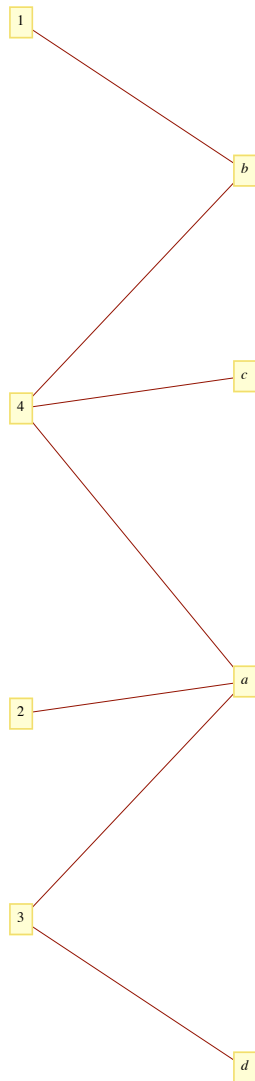
```
In[1]:= bipartite = {1 → b, 2 → a, 3 → a, 3 → d, 4 → c, 4 → a, 4 → b};
g = Join[bipartite,
  Map[{Left → #[[1]]} &, bipartite], Map[{Right → #[[2]]} &, bipartite]];
In[3]:= GraphPlot[g,
  VertexCoordinateRules → {left → {-2, 0}, right → {2, 0}, Sequence@@Flatten[
    Map[({#[[1]} → {-1, Automatic}, #[[2]} → {1, Automatic})] &, bipartite]}},
  VertexLabeling → True, VertexRenderingFunction →
  (If[#2 != Left && #2 != Right, Text[#2, #1, Background → White], {}] &),
  EdgeRenderingFunction → (If[! MemberQ[{Left, Right}, #2[[1]]] &&
    ! MemberQ[{Left, Right}, #2[[2]]], Line[#1], {}] &)]
```



When the bipartite graph is connected, it works even better without augmenting it with "Left" and "Right" anchors.

```
In[4]:= GraphPlot[bipartite, VertexCoordinateRules →  
  Flatten[Map[({#[[1]] → {-1, Automatic}, #[[2]] → {1, Automatic}}) &, bipartite]],  
  VertexLabeling → True]
```

Out[4]=

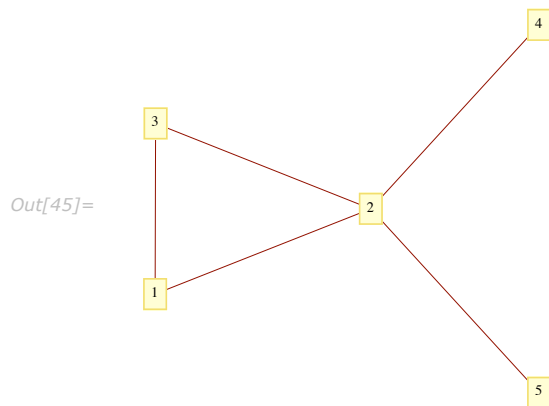


VertexLabeling

The option `VertexLabeling` specifies whether to show vertex names as labels. Possible values for this option are `True`, `False`, `Automatic` (the default) and `Tooltip`. `VertexLabeling -> True` shows the labels. For graphs specified by an adjacency matrix, vertex labels are taken to be successive integers $1, 2, \dots, n$, where n is the size of the matrix. For graphs specified by a list of rules, labels are the expressions used in the rules. `VertexLabeling -> False` displays each vertex as a point. `VertexLabeling -> Tooltip` displays each vertex as a point, but gives its name in a tooltip. `VertexLabeling -> Automatic` displays each vertex as a point, giving its name in a tooltip if the number of vertices is not too large. You can also use `Tooltip[vk, vlbl]` anywhere in the list of rules to specify an alternative tooltip for a vertex v_k .

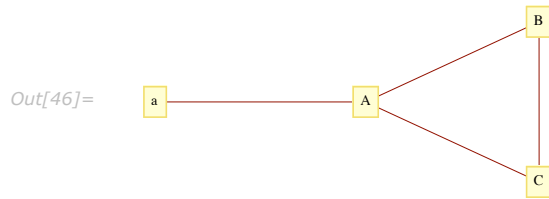
This draws the graph with labels given as indices of the adjacency matrix.

```
In[45]:= GraphPlot[ $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$ , VertexLabeling -> True]
```



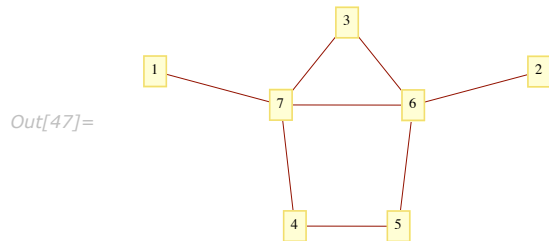
This uses the labels specified in the list of rules.

```
In[46]:= GraphPlot[{"A" → "B", "A" → "a", "B" → "C", "C" → "A"}, VertexLabeling → True]
```



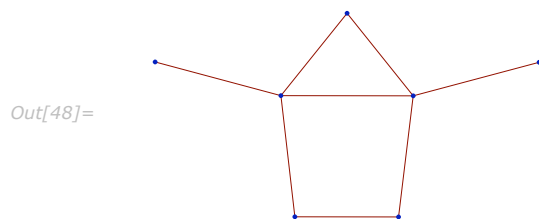
This specifies alternative labels for vertices 3 and 5. Place the cursor above the vertices to see the labels.

```
In[47]:= GraphPlot[{5 → 4, 6 → 2, 6 → Tooltip[3, "number 3"], 6 → 5, 7 → 1, 7 → 3, 7 → 4, 7 → 6}, VertexLabeling → True]
```



This plots vertices as points, and displays vertex names in tooltips. Place the cursor above the vertices to see the labels.

```
In[48]:= GraphPlot[{5 → 4, 6 → 2, 6 → 3, 6 → 5, 7 → 1, 7 → 3, 7 → 4, 7 → 6}, VertexLabeling → Tooltip]
```



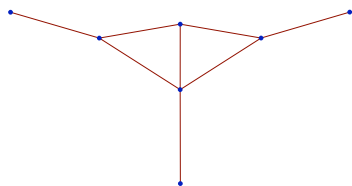
VertexRenderingFunction

The option `VertexRenderingFunction` specifies graphical representation of the graph edges. Possible values for this option are `Automatic`, `None`, or a function that gives a proper combination of graphics primitives and directives. With the default setting of `Automatic`, vertices are displayed as points, with their names given in tooltips.

By default, vertices are displayed as points and, for small graphs, labeled in tooltips. Point the cursor at a vertex to see the tooltip.

```
In[49]:= GraphPlot[{5 → 3, 5 → 4, 6 → 2, 6 → 4, 7 → 1, 7 → 4, 7 → 5, 7 → 6}]
```

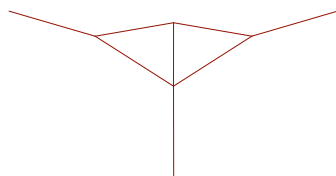
Out[49]=



This draws no vertices at all.

```
In[50]:= GraphPlot[{5 → 3, 5 → 4, 6 → 2, 6 → 4, 7 → 1, 7 → 4, 7 → 5, 7 → 6},  
VertexRenderingFunction → None]
```

Out[50]=

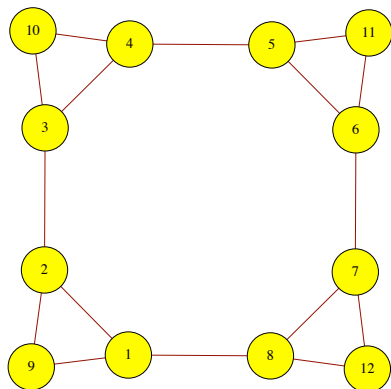


With `VertexRenderingFunction -> g`, each vertex is rendered with the graphics primitives given by $g[r_i, v_i, \dots]$, where r_i is the coordinate of the vertex and v_i is the label of the vertex. Explicit settings for `VertexRenderingFunction -> g` override settings for `VertexLabeling`.


This shows vertices as yellow disks.

```
In[51]:= GraphPlot[{1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 7, 7 → 8, 8 → 1, 1 → 9, 2 → 9,  
3 → 10, 4 → 10, 6 → 11, 5 → 11, 7 → 12, 8 → 12}, VertexRenderingFunction →  
({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]
```

Out[51]=

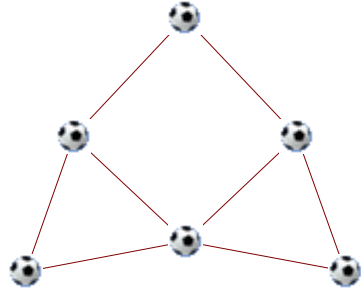


This renders vertices using a predefined graphic.

```
In[52]:= gr = ;
```

```
In[53]:= GraphPlot[{4 → 2, 4 → 3, 5 → 1, 5 → 3, 6 → 1, 6 → 2, 6 → 4, 6 → 5},
  VertexRenderingFunction → (Inset[gr, #1] &)]
```

Out[53]=



A Common Suboption of All Methods

All graph drawing methods accept the method suboption "Rotation", which specifies the desired amount of clockwise rotation in radians from the default orientation. The option takes any numeric values, or `False`. The default is 0.

For `GraphPlot` and `GraphPlot3D`, the default orientation is derived by an alignment step where the principal axis is found and the graph drawing is aligned with the x coordinate. However if "Rotation" \rightarrow `False` is specified, this step is skipped.

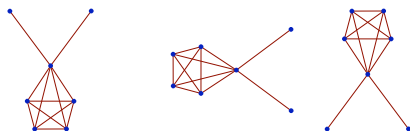
<i>option name</i>	<i>default value</i>	
"Rotation"	0	amount of clockwise rotation to apply to the drawing

A common suboption for all methods.

This rotates a plot of a graph by $\pi/2$, 0, and $-\pi/2$ clockwise.

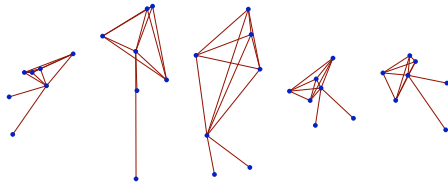
```
In[3]:= GraphicsRow[Table[GraphPlot[
  {4 → 3, 5 → 3, 5 → 4, 6 → 3, 6 → 4, 6 → 5, 7 → 1, 7 → 2, 7 → 3, 7 → 4, 7 → 5, 7 → 6},
  Method → {"Automatic", "Rotation" → rot}], {rot, {-Pi/2, 0, Pi/2}}]]
```

Out[2]=

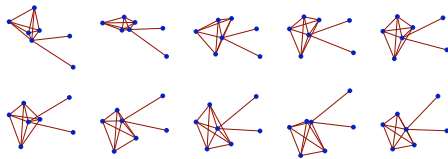


This shows the evolution of a graph layout process.

```
In[4]:= Grid[Partition[
  Table[GraphPlot[{4 → 3, 5 → 3, 5 → 4, 6 → 3, 6 → 4, 6 → 5, 7 → 1, 7 → 2, 7 → 3, 7 → 4,
    7 → 5, 7 → 6}, Method → {"SpringElectricalEmbedding", "Rotation" → False,
    "RecursionMethod" → None, MaxIterations → i}, ImageSize → 50], {i, 15}], {5}]]
```



Out[1]=



Common Suboptions of the "SpringEmbedding" and "SpringElectricalEmbedding" Methods

Both the `SpringEmbedding` and `SpringElectricalEmbedding` methods belong to the family of so-called force-directed methods. These methods work by calculating the force on each vertex, and iteratively moving the vertex along the force in an effort to minimize the overall system's energy. See [8] for algorithmic details. These two methods have the following common options.

<i>option name</i>	<i>default value</i>	
"EnergyControl"	Automatic	how the energy function is controlled during minimization
"InferentialDistance"	Automatic	cutoff distance beyond which the force calculation ignores inference from faraway vertices
MaxIterations	Automatic	maximum number of iterations to be used in attempting to minimize the energy
"RandomSeed"	Automatic	seed to use in the random generator for initial vertex placement
"RecursionMethod"	Automatic	whether a multilevel algorithm is used to lay out the graph

"StepControl"	Automatic	how step lengths are modified during energy minimization
"StepLength"	Automatic	initial step length used in moving the vertices
"Tolerance"	Automatic	tolerance used in terminating the energy minimization process

Common suboptions for "SpringEmbedding" and "SpringElectricalEmbedding" methods.

"EnergyControl"

The suboption "EnergyControl" specifies limitations on the total energy of the system during minimization. Possible values are Automatic (the default), "Monotonic", or "NonMonotonic". When the value is "Monotonic", a step along the force will only be accepted if the energy is lowered. When the value is "NonMonotonic", a step along the force will be accepted even if the energy is not lowered.

"InferentialDistance"

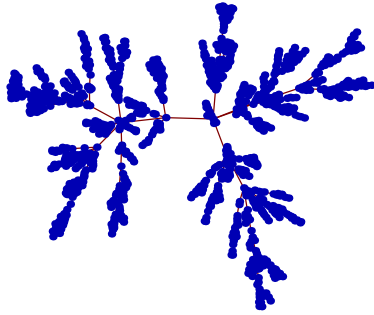
The suboption "InferentialDistance" specifies a cutoff distance beyond which the interaction between vertices is assumed to be nonexistent. Possible values are Automatic (the default) or a positive numeric value. For the "SpringEmbedding" method, if the graph distance between a vertex i and a vertex j is greater than the option value of "InferentialDistance", the repulsive and attractive spring force between i and j is ignored. For the "SpringElectricalEmbedding" method, if the Euclidean distance between a vertex i and a vertex j is greater than the option value of "InferentialDistance", the repulsive force between i and j is ignored.

This draws a random tree using the "SpringElectricalEmbedding" method.

```
In[1]:= g = RandomInteger[#] → # + 1 & /@ Range[0, 1000];
```

```
In[2]:= GraphPlot[g, Method -> "SpringElectricalEmbedding"]
```

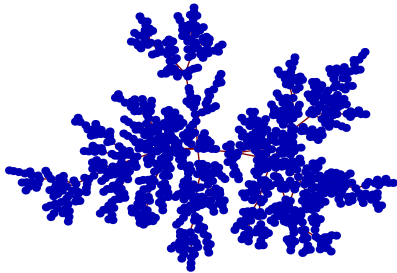
Out[2]=



Using a smaller (more negative) "RepulsiveForcePower" option value (see the next section), the graph now fills more space.

```
In[22]:= GraphPlot[g, Method -> {"SpringElectricalEmbedding", "RepulsiveForcePower" -> -2}]
```

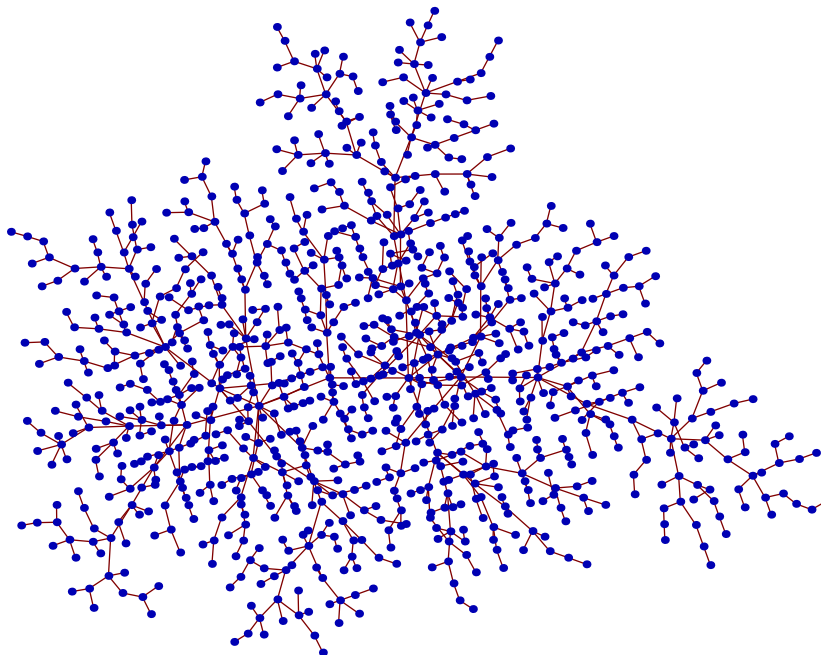
Out[22]=



A similar effect can be achieved using a small "InferentialDistance" option value.

```
In[21]:= GraphPlot[g, Method -> {"SpringElectricalEmbedding", "InferentialDistance" -> .5}]
```

Out[21]=



MaxIterations

The option `MaxIterations` specifies the maximum number of iterations to be used in attempting to minimize the energy. Possible values are `Automatic` (the default) or a positive integer.

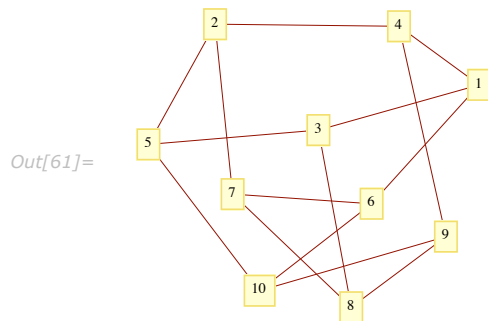
"RandomSeed"

The option `"RandomSeed"` specifies a seed for the random number generator that computes the initial vertex placement. Changing this option usually affects the orientation of the drawing of the graph, but it can also change the layout. Possible values are `Automatic` or an integer.

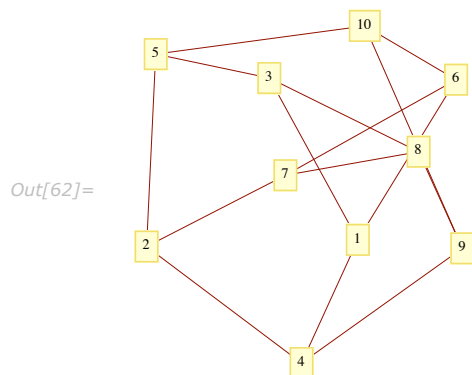
This shows the effect of different random seed values on drawing the Petersen graph.

```
In[60]:= g = {1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7,
             7 → 8, 8 → 9, 9 → 10, 6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10};
```

```
In[61]:= GraphPlot[g, Method → {"SpringElectricalEmbedding", "RandomSeed" → Automatic},
           VertexLabeling → True]
```



```
In[62]:= GraphPlot[g, Method → {"SpringElectricalEmbedding", "RandomSeed" → 4321},
           VertexLabeling → True]
```



"RecursionMethod"

The option "RecursionMethod" specifies whether the graph layout should be produced by a recursive procedure. Possible values are `Automatic` (the default), `"Multilevel"`, or `None`. In a `"Multilevel"` algorithm, the graph is successively coarsened into graphs with a smaller and smaller number of vertices. The coarser graphs are laid out first, and those layouts are interpolated into the finer graphs and then further refined.

<i>suboption name</i>	<i>default value</i>	
"Randomization"	<code>Automatic</code>	whether to inspect vertices in random order
"MinSize"	<code>Automatic</code>	minimal number of vertices in a coarsened graph
"CoarseningScheme"	<code>Automatic</code>	how graphs are coarsened

Suboptions for `"Multilevel"`.

For the option `"Randomize"`, possible values are `Automatic`, `True`, and `False`. For `"MinSize"`, possible values are `Automatic` or a positive number. For `"CoarseningScheme"`, the implemented algorithms are based on either a maximal independent vertex set, which forms the coarse vertices, or a maximal independent edge set, also called a matching. In a matching, two vertices that form an edge are merged to form a coarse graph vertex. The following are possible values for `"CoarseningScheme"`.

"MaximalIndependentVertexSet"	link vertices in the maximal independent set if their graph distance is 3 or less
"MaximalIndependentVertexSetInjection"	link vertices in the maximal independent set if their graph distance is 1 or 2
"MaximalIndependentVertexSetRugeStuben"	generate the maximal independent vertex set, giving priority to vertices with more neighbors not in the set, then link vertices in the set if their graph distance is 3 or less
"MaximalIndependentVertexSetRugeStubenInjection"	link vertices if their graph distance is 1 or 2, giving priority to vertices with more neighbors
"MaximalIndependentEdgeSet"	consider edges in their natural order when matching

"MaximalIndependentEdgeSetHeavyEdge"

give priority to edges with higher edge weight (i.e., edges that represent a larger number of edges in the original graph) when matching

"MaximalIndependentEdgeSetSmallestVertexWeight"

give priority to matchings of vertices with neighbors that have the smallest vertex weight

"StepControl"

The option "StepControl" defines how step length is modified during energy minimization. It can be `Automatic` (the default), `Monotonic` (where step length can only be decreased), `NonMonotonic` (where step length can be made larger or smaller), or `StrictlyMonotonic` (where step length is strictly reduced between iterations).

"StepLength"

The option "StepLength" gives the initial step length used in moving the vertices around. Possible values are `Automatic` (the default) or a positive real number.

Tolerance

The option `Tolerance` specifies the tolerance used in terminating the energy minimization process. If the average change of coordinates of each vertex is less than the tolerance, the energy minimization process is terminated and the current coordinates are given as output. Possible values are `Automatic` or a positive real number.

Method Suboptions of the "SpringElectricalEmbedding" Method

<i>option name</i>	<i>value</i>	
"Octree"	Automatic	whether to use an octree data structure (in three dimensions) or a quadtree data structure (in two dimensions) in the calculation of repulsive force
"RepulsiveForcePower"	-1	how fast the repulsive force decays over distance

Method options for "SpringElectricalEmbedding".

"Octree"

The "octree" option specifies whether to use an octree data structure (in three dimensions) or a quadtree data structure (in two dimensions) in the calculation of repulsive force. Possible values are `Automatic` (the default), `True`, or `False`. Use of an octree/quadtree data structure minimizes the complexity of computation by approximating the long-range repulsive force. However, it introduces an approximation to the force calculation. Therefore, in a few cases the result may not be as good.

"RepulsiveForcePower"

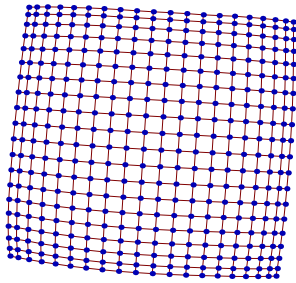
Possible values are negative real numbers, with -1 as the default. In the spring-electrical embedding, the repulsive force between two vertices i and j is K^2/d_{ij} by default. If the value of `RepulsiveForcePower` is r (with $r < 0$), then the repulsive force is defined as $K^{1-r} d_{ij}^r$, where d_{ij} is the distance between the vertices and K is a constant coefficient.

A strong long-range repulsive force over long distance often has the boundary effect that vertices in the periphery are closer to each other than those in the center are. Specifying a weaker long-range repulsive force can sometimes alleviate this effect. This option can also be useful in drawing a graph so that it fills up more space. (See the "InferentialDistance" method option for details.)

With a repulsive force power of -2, the boundary vertices are not as close to each other as they are with the default value of -1.

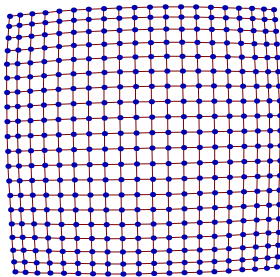
```
In[63]:= GraphPlot[GridGraph[20, 20], Method → "SpringElectricalEmbedding"]
```

Out[63]=



```
In[64]:= GraphPlot[GridGraph[20, 20],  
Method → {"SpringElectricalEmbedding", "RepulsiveForcePower" → -2}]
```

Out[64]=



Method Suboption of "HighDimensionalEmbedding"

<i>option name</i>	<i>default value</i>	
"RefinementMethod"	None	whether the result should be further refined, and which method should be used for refinement

Method option for "HighDimensionalEmbedding".

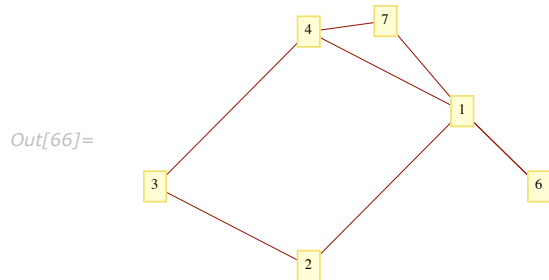
"RefinementMethod"

The option "RefinementMethod" specifies whether the result should be further refined, and which method should be used to refine it. Possible values are `None` (the default), `"SpringEmbedding"`, or `"SpringElectricalEmbedding"`.

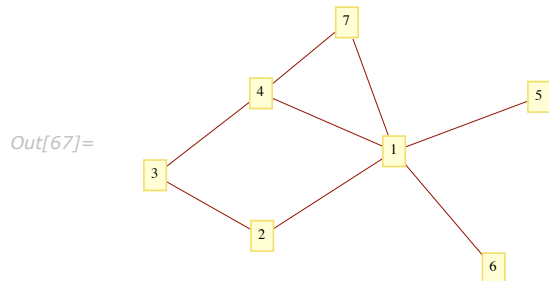
This shows a case where the "HighDimensionalEmbedding" method placed vertices 5 and 6 at the same position. Specifying a "RefinementMethod" option helps to draw the graph better.

```
In[65]:= g = {1 → 2, 2 → 3, 3 → 4, 4 → 1, 1 → 5, 1 → 6, 1 → 7, 4 → 7};
```

```
In[66]:= GraphPlot[g, Method -> "HighDimensionalEmbedding", VertexLabeling -> True]
```



```
In[67]:= GraphPlot[g, Method -> {"HighDimensionalEmbedding",  
"RefinementMethod" -> "SpringElectricalEmbedding"}, VertexLabeling -> True]
```



Advanced Topics

Drawing a Graph to Fill Up More Space

While the default setting for `GraphPlot` works well in general, for graphs that have a wide range of values for the vertex degree, it is often necessary to use a setting that helps the vertices to occupy less space.

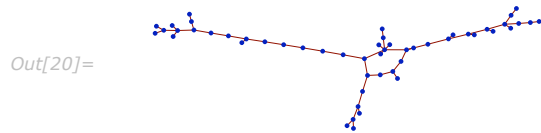
The default method usually works well.

```
In[18]:= SeedRandom[321];
```

```
In[19]:= g = Table[i → RandomInteger[{1, 52}], {i, 52}]
```

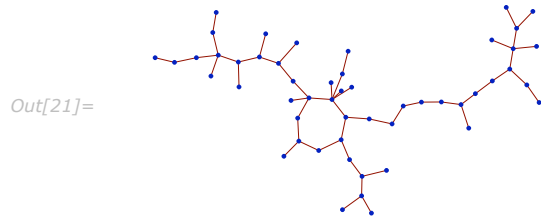
```
Out[19]= {1 → 11, 2 → 27, 3 → 15, 4 → 18, 5 → 11, 6 → 44, 7 → 11, 8 → 50, 9 → 51, 10 → 6, 11 → 52,
12 → 31, 13 → 23, 14 → 50, 15 → 37, 16 → 6, 17 → 31, 18 → 19, 19 → 40, 20 → 21, 21 → 43,
22 → 49, 23 → 11, 24 → 16, 25 → 27, 26 → 46, 27 → 16, 28 → 4, 29 → 11, 30 → 17, 31 → 3,
32 → 25, 33 → 44, 34 → 51, 35 → 27, 36 → 29, 37 → 22, 38 → 42, 39 → 8, 40 → 21, 41 → 46,
42 → 2, 43 → 23, 44 → 45, 45 → 23, 46 → 4, 47 → 14, 48 → 8, 49 → 52, 50 → 30, 51 → 8, 52 → 19}
```

```
In[20]:= GraphPlot[g]
```



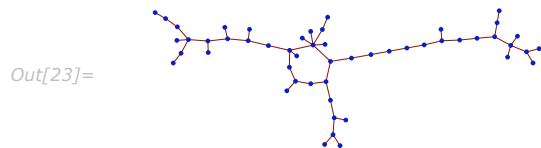
However, sometimes "SpringEmbedding" produces a drawing that occupies more space.

```
In[21]:= GraphPlot[g, Method → "SpringEmbedding"]
```



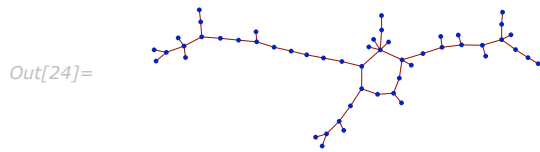
A similar effect can be achieved with a repulsive force power smaller than the default (-1), so that the repulsive force decays more quickly over distance.

```
In[23]:= GraphPlot[g, Method → {"SpringElectricalEmbedding", "RepulsiveForcePower" → -1.8}]
```



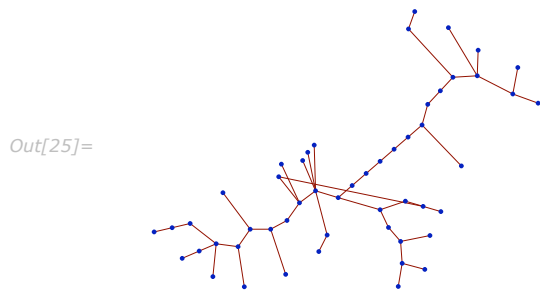
Alternatively, specify a cutoff distance.

```
In[24]:= GraphPlot[g, Method → {"SpringElectricalEmbedding", "InferentialDistance" → 4}]
```



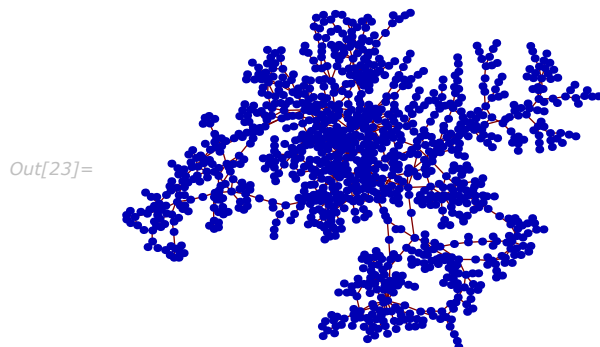
For such tree-like graphs, a tree-drawing algorithm may be preferable. See "Tree Drawing" for greater control over tree layout.

```
In[25]:= GraphPlot[g, Method → "RadialDrawing"]
```



This draws a graph from power network modeling.

```
In[23]:= GraphPlot[ExampleData[{"Matrix", "HB/1138_bus"}, "Matrix"],
  Method → {"SpringElectricalEmbedding", "InferentialDistance" → 4,
    "RepulsiveForcePower" → -1.8}, PlotRangePadding → 0]
```



Improving Performance for Drawing Very Large Graphs

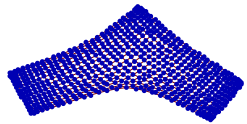
Although the default option set usually ensures a very good performance, it is often possible to further increase drawing speed and reduce memory usage by selecting specific option combinations for a particular task. For example, speed and/or memory usage can be improved using a smaller number of iterations, a smaller inferential distance, or a lower tolerance. These settings tend to reduce quality, but still frequently offer an acceptable compromise.

This is a drawing with default option settings.

```
In[75]:= g = Import["LinearAlgebraExamples/Data/nos6.mtx"];
```

```
In[76]:= GraphPlot[g] // Timing
```

```
Out[76]= {0.325451, {
```

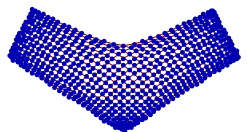


```
}]
```

A coarsening scheme based on the maximal independent vertex set is often faster and uses less memory, and yet offers a comparable layout quality.

```
In[77]:= GraphPlot[g,
  Method → {"SpringElectricalEmbedding", "RecursionMethod" → {"Multilevel",
    "CoarseningScheme" → "MaximalIndependentVertexSetRugeStuben"}}] // Timing
```

```
Out[77]= {0.270455, {
```

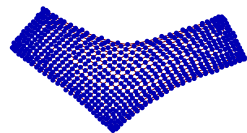


```
}]
```

By reducing the number of iterations to 30, you can get a result still faster.

```
In[78]:= GraphPlot[g, Method → {"SpringElectricalEmbedding", MaxIterations → 30}] // Timing
```

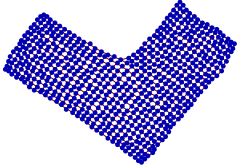
```
Out[78]= {0.154644, {
```



```
}]
```

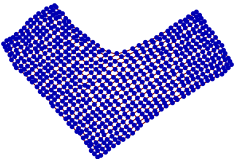
Setting the inferential distance to 2 and the number of iterations to 40 is also faster than the default.

```
In[79]:= GraphPlot[g, Method → {"SpringElectricalEmbedding",
  "InferentialDistance" → 2, MaxIterations → 40}] // Timing
```

```
Out[79]= {0.17386, 
```

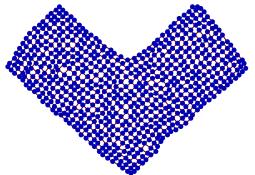
By further reducing the number of iterations to 20, you get a result much faster.

```
In[80]:= GraphPlot[g, Method → {"SpringElectricalEmbedding",
  "InferentialDistance" → 2, MaxIterations → 20}] // Timing
```

```
Out[80]= {0.090956, 
```

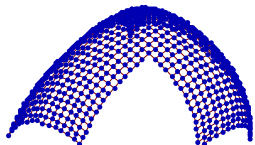
A combination of the previous options is still faster.

```
In[81]:= GraphPlot[g, Method → {"SpringElectricalEmbedding", "InferentialDistance" → 2,
  MaxIterations → 20, "RecursionMethod" → {"Multilevel",
  "CoarseningScheme" → "MaximalIndependentVertexSetRugeStuben"}}] // Timing
```

```
Out[81]= {0.076418, 
```

"HighDimensionalEmbedding" tends to be the fastest method, but the quality of the drawing often suffers.

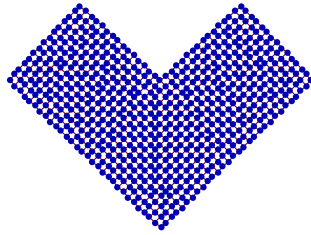
```
In[82]:= GraphPlot[g, Method → "HighDimensionalEmbedding"] // Timing
```

```
Out[82]= {0.013691, 
```

For comparison, "SpringEmbedding" is the slowest method, but it is the only one that draws the mesh using orthogonal lines.

```
In[83]:= GraphPlot[g, Method -> "SpringEmbedding"] // Timing
```

```
Out[83]= {1.47127, }
```



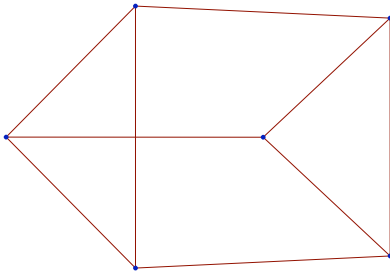
Extracting Vertex Coordinates from Output

In most cases, you will deal with the output of `GraphPlot` and `GraphPlot3D` just as with a usual `Graphics` expression. However, you may sometimes want to take advantage of the additional information encapsulated in the output expression, which has the form `Graphics[Annotation[data, VertexCoordinateRules -> rules]]`. Particularly, it is sometimes useful to extract coordinates of graph vertices.

Here is a simple graph.

```
In[84]:= GraphPlot[{3 -> 2, 4 -> 1, 4 -> 3, 5 -> 1, 5 -> 2, 5 -> 4, 6 -> 1, 6 -> 2, 6 -> 3}]
```

```
Out[84]=
```



This extracts the coordinates of vertices.

```
In[85]:= VertexCoordinateRules /. Cases[%, _Rule, Infinity]
```

```
Out[85]= {{0.578783, 1.17222}, {0.578837, 0.}, {1.71796, 1.11836},
          {1.15021, 0.585401}, {1.71821, 0.0540483}, {0., 0.585946}}
```


Example Gallery

E. coli Transcription Networks

In a graph representation of a transcriptional regulation network that controls gene expression in cells, nodes (vertices) are operons, which are one or more genes transcribed on the same messenger ribonucleic acid (mRNA). Edges of the graph are directed from an operon that encodes a transcription factor to an operon that it directly regulates [1].

Data

This is the network [2] described as rules.

```
In[86]:= g = {"acrR" → "acrAB", 2}, {"ada_alkB" → "aidB", 1}, {"ada_alkB" → "alkA", 1},
{"adiA_adiY" → "adiA", 1}, {"alpA" → "slp", 1}, {"appY" → "appCBA", 1},
{"araC" → "araBAD", 3}, {"araC" → "araE", 3}, {"araC" → "araFG_araH_1H_2", 3},
{"araC" → "araJ", 1}, {"araC" → "aceBAK", 2}, {"araC" → "appCBA", 1},
{"araC" → "appY", 1}, {"araC" → "betIBA", 2}, {"araC" → "cydAB", 1},
{"araC" → "cyoABCDE", 2}, {"araC" → "dctA", 2}, {"araC" → "fadBA", 2},
{"araC" → "focA_pflB", 1}, {"araC" → "fumA", 2}, {"araC" → "fumC", 2},
{"araC" → "glcDEFGB", 2}, {"araC" → "glpACB", 2}, {"araC" → "gltA", 2},
{"araC" → "icdA", 2}, {"araC" → "lctPRD", 2}, {"araC" → "mdh", 2},
{"araC" → "nuoABCEFGHIJKLMN", 2}, {"araC" → "sdhCDAB_b0725_sucABCD", 2},
{"araC" → "sodA", 2}, {"argR" → "argCBH", 2}, {"argR" → "argD", 2},
{"argR" → "argE", 2}, {"argR" → "argF", 2}, {"argR" → "argI", 2},
{"argR" → "carAB", 2}, {"arsR" → "arseFG", 2}, {"asnC" → "asna", 1},
{"atoc" → "atoDAB", 1}, {"atoc" → "atoDAE", 1}, {"betIBA" → "betT", 2},
{"birA_murA" → "bioA", 2}, {"birA_murA" → "bioBFCD", 2}, {"cadC" → "cadBA", 1},
{"caiF" → "caiABCDE", 1}, {"caiF" → "fixABCX", 1}, {"cbl" → "ssuEADCB", 1},
{"cbl" → "tauABCD", 1}, {"cpxAR" → "cpxP", 1}, {"cpxAR" → "dsbA", 1},
{"cpxAR" → "ecfI", 1}, {"cpxAR" → "htrA", 1}, {"cpxAR" → "motABcheAW", 2},
{"cpxAR" → "rotA", 1}, {"cpxAR" → "skp_lpxDA_fabZ", 1}, {"cpxAR" → "tsr", 2},
{"cpxAR" → "xprB_dsbC_recJ", 1}, {"crp" → "acs", 1}, {"crp" → "aldB", 1},
{"crp" → "ansB", 1}, {"crp" → "araBAD", 1}, {"crp" → "araC", 1},
{"crp" → "araE", 1}, {"crp" → "araFG_araH_1H_2", 1}, {"crp" → "araJ", 1},
{"crp" → "caiF", 1}, {"crp" → "caiABCDE", 1}, {"crp" → "cirA", 1},
{"crp" → "cpdB", 1}, {"crp" → "cyAA", 2}, {"crp" → "dadAX", 1},
{"crp" → "dctA", 2}, {"crp" → "dcuB_fumB", 2}, {"crp" → "deoCABD", 3},
{"crp" → "dsdXA", 1}, {"crp" → "ebgAC", 1}, {"crp" → "epd_pgk", 1},
{"crp" → "fadL", 2}, {"crp" → "fixABCX", 1}, {"crp" → "flhDC", 1},
{"crp" → "focA_pflB", 1}, {"crp" → "fucAO", 1}, {"crp" → "fucPIKUR", 1},
{"crp" → "fur", 1}, {"crp" → "galETKM", 3}, {"crp" → "galS", 1},
{"crp" → "glgCAP", 1}, {"crp" → "glgS", 1}, {"crp" → "glnALG", 1},
{"crp" → "glpACB", 1}, {"crp" → "glpD", 1}, {"crp" → "glpFK", 1},
{"crp" → "glpTQ", 1}, {"crp" → "gltA", 1}, {"crp" → "gntKU", 1},
{"crp" → "gntT", 1}, {"crp" → "ivbL_ilvBN", 1}, {"crp" → "lacZYA", 1},
{"crp" → "malEFG", 1}, {"crp" → "malI", 2}, {"crp" → "malK_lamB_malM", 1},
{"crp" → "malS", 1}, {"crp" → "malT", 1}, {"crp" → "malXY", 3},
{"crp" → "manXYZ", 1}, {"crp" → "melAB", 1}, {"crp" → "melR", 1},
{"crp" → "mglBAC", 1}, {"crp" → "nagBACD", 1}, {"crp" → "nagE", 1},
{"crp" → "nupG", 3}, {"crp" → "ompA", 2}, {"crp" → "ppiA", 3}, {"crp" → "proP", 3},
{"crp" → "ptsHI_crr", 3}, {"crp" → "rhaBAD", 3}, {"crp" → "rhaT", 1},
```

```

{"crp" → "rpoH", 3}, {"crp" → "sdhCDAB_b0725_sucABCD", 1}, {"crp" → "speC", 2},
{"crp" → "srlAEBD_gutM_srlR_gutQ", 1}, {"crp" → "tdcABCDEFGF", 1},
{"crp" → "tnaLAB", 1}, {"crp" → "tsx", 3}, {"crp" → "ubiG", 1}, {"crp" → "udp", 1},
{"crp" → "uhpT", 1}, {"crp" → "yhFA", 3}, {"crp" → "yiaKLMNOPQRS", 1},
{"csgDEFG" → "csgBA", 1}, {"cspA" → "gyrA", 1}, {"cspA" → "hns", 1},
{"cynR" → "cynTSX", 1}, {"cysB" → "cysDNC", 1}, {"cysB" → "cysJIH", 1},
{"cysB" → "cysK", 1}, {"cysB" → "cysPUWAM", 1}, {"cysB" → "tauABCD", 1},
{"cytR" → "deoCABD", 2}, {"cytR" → "nupC", 2}, {"cytR" → "nupG", 2},
{"cytR" → "ppiA", 2}, {"cytR" → "rpoH", 2}, {"cytR" → "tsx", 3},
{"cytR" → "udp", 2}, {"deoR" → "deoCABD", 2}, {"deoR" → "nupG", 2},
{"deoR" → "tsx", 2}, {"dnaA" → "nrdAB", 1}, {"dnaA" → "rpoH", 2},
{"dsdC" → "dsdXA", 1}, {"ebgR" → "ebgAC", 2}, {"envY_ompT" → "ompC", 1},
{"envY_ompT" → "ompF", 1}, {"evgA" → "ompC", 1}, {"exuR" → "exuT", 2},
{"exuR" → "uxaCA", 2}, {"exuR" → "uxuABR", 2}, {"fadR" → "fabA", 1},
{"fadR" → "fadBA", 2}, {"fadR" → "fadL", 2}, {"fadR" → "iclMR", 2},
{"fadR" → "usPA", 2}, {"fecIR" → "fecABCDE", 1}, {"fhla" → "fdhF", 1},
{"fhla" → "hycABCDEFGH", 1}, {"fhla" → "hypABCDE", 1}, {"flhDC" → "flgAMN", 1},
{"flhDC" → "flgBCDEFGHIJK", 1}, {"flhDC" → "flhBAE", 1}, {"flhDC" → "fliAZY", 1},
{"flhDC" → "fliE", 1}, {"flhDC" → "fliFGHIJK", 1}, {"flhDC" → "fliLMNOPQR", 1},
{"fliAZY" → "flgBCDEFGHIJK", 1}, {"fliAZY" → "flgKL", 1},
{"fliAZY" → "flgMN", 1}, {"fliAZY" → "flhBAE", 1}, {"fliAZY" → "fliC", 1},
{"fliAZY" → "fliDST", 1}, {"fliAZY" → "fliE", 1}, {"fliAZY" → "fliFGHIJK", 1},
{"fliAZY" → "fliLMNOPQR", 1}, {"fliAZY" → "motABcheAW", 1},
{"fliAZY" → "tarTapcheRBYZ", 1}, {"fliAZY" → "tsr", 1}, {"fnr" → "acs", 1},
{"fnr" → "ansB", 1}, {"fnr" → "arCA", 1}, {"fnr" → "asPA", 1},
{"fnr" → "caif", 1}, {"fnr" → "cydAB", 2}, {"fnr" → "cyoABCDE", 2},
{"fnr" → "dcuB_fumB", 1}, {"fnr" → "dmsABC", 1}, {"fnr" → "fdnGHI", 1},
{"fnr" → "focA_pflB", 1}, {"fnr" → "frdABCD", 1}, {"fnr" → "glpACB", 1},
{"fnr" → "hypABCDE", 1}, {"fnr" → "icdA", 2}, {"fnr" → "narGHJI", 1},
{"fnr" → "narK", 1}, {"fnr" → "ndh", 2}, {"fnr" → "nirBDC_cysG", 1},
{"fnr" → "nuoABCEFGHIJKLMN", 2}, {"fnr" → "sdhCDAB_b0725_sucABCD", 2},
{"fnr" → "tdcABCDEFGF", 1}, {"FruR" → "aceBAK", 2}, {"FruR" → "adHE", 2},
{"FruR" → "fruBKA", 2}, {"FruR" → "icdA", 1}, {"FruR" → "ppsA", 1},
{"FruR" → "ptsHI_crr", 3}, {"FruR" → "pykF", 2}, {"fucPIKUR" → "fucAO", 1},
{"fur" → "cirA", 2}, {"fur" → "entCEBA", 2}, {"fur" → "fecIR", 2},
{"fur" → "fepA_entD", 2}, {"fur" → "fepB", 2}, {"fur" → "fepDGC", 2},
{"fur" → "fhuACDB", 2}, {"fur" → "soda", 2}, {"fur" → "tonB", 2},
{"GalR" → "gaLETKM", 2}, {"GalR" → "gals", 2}, {"gals" → "mglBAC", 2},
{"gatR_1" → "gatYZABCD", 2}, {"gcvA" → "gcvTHP", 3}, {"gcvR" → "gcvTHP", 2},
{"glcC" → "glcDEFG", 1}, {"glnALG" → "glnHPQ", 3}, {"glnALG" → "nac", 1},
{"glpR" → "glpACB", 2}, {"glpR" → "glpD", 2}, {"glpR" → "glpFK", 2},
{"glpR" → "glpTQ", 2}, {"gntR" → "edd_eda", 2}, {"gntR" → "gntKU", 2},
{"gntR" → "gntT", 2}, {"hcaR" → "hcaA1A2CBD_yphA", 1}, {"hima" → "aceBAK", 1},
{"hima" → "caiTABCDE", 2}, {"hima" → "carAB", 3}, {"hima" → "dps", 1},
{"hima" → "ecpD_htrE", 1}, {"hima" → "focA_pflB", 1}, {"hima" → "glcDEFG", 1},
{"hima" → "glnHPQ", 3}, {"hima" → "himD", 2}, {"hima" → "hycABCDEFGH", 1},
{"hima" → "hypABCDE", 1}, {"hima" → "narGHJI", 1}, {"hima" → "narK", 1},
{"hima" → "nuoABCEFGHIJKLMN", 2}, {"hima" → "ompC", 2}, {"hima" → "ompF", 3},
{"hima" → "ompR_envZ", 2}, {"hima" → "osmE", 2}, {"hima" → "pspABCDE", 1},
{"hima" → "soda", 2}, {"hima" → "tdcABCDEFGF", 1}, {"hns" → "caif", 2},
{"hns" → "flhDC", 1}, {"hns" → "fliAZY", 1}, {"hns" → "nhaA", 1},
{"hns" → "osmC", 1}, {"hns" → "rcsAB", 1}, {"hns" → "stpA", 2},
{"hydHG" → "zraP", 1}, {"iclMR" → "aceBAK", 2}, {"iclMR" → "acs", 2},
{"ilvY" → "ilvC", 1}, {"kdpDE" → "kdpABC", 1}, {"lacI" → "lacZYA", 2},
{"leuO" → "leuLABCD", 1}, {"lexA_dinF" → "polB", 2}, {"lexA_dinF" → "recA", 2},
{"lexA_dinF" → "recN", 2}, {"lexA_dinF" → "rpsU_dnaG_rpod", 2},
{"lexA_dinF" → "ssb", 2}, {"lexA_dinF" → "sula", 2}, {"lexA_dinF" → "umuDC", 2},
{"lexA_dinF" → "uvrA", 2}, {"lexA_dinF" → "uvrB", 2}, {"lexA_dinF" → "uvrC", 2},
{"lexA_dinF" → "uvrD", 2}, {"lrp" → "gcvTHP", 1}, {"lrp" → "gltBDF", 1},
{"lrp" → "ilvIH", 1}, {"lrp" → "kbl_tdh", 3}, {"lrp" → "livJ", 2},
{"lrp" → "livKHMgf", 2}, {"lrp" → "lysU", 2}, {"lrp" → "ompC", 2},
{"lrp" → "ompF", 1}, {"lrp" → "oppABCD", 2}, {"lrp" → "osmC", 1},
{"lrp" → "sdaA", 2}, {"lrp" → "serA", 1}, {"lrp" → "stpA", 1},

```

```

{"lysR" → "lysA", 1}, {"lysR" → "tdcABCDEFG", 1}, {"malI" → "malXY", 2},
{"malT" → "maleFG", 1}, {"malT" → "malK_lamB_malM", 1}, {"malT" → "malPQ", 1},
{"malT" → "malS", 1}, {"malT" → "malZ", 1}, {"marRAB" → "fpr", 1},
{"marRAB" → "fumC", 1}, {"marRAB" → "nfo", 1}, {"marRAB" → "soda", 1},
{"marRAB" → "zwf", 1}, {"melR" → "melAB", 1}, {"metJ" → "metA", 2},
{"metJ" → "metC", 2}, {"metJ" → "metF", 2}, {"metJ" → "metR", 2},
{"metR" → "glyA", 1}, {"metR" → "metA", 1}, {"metR" → "meth", 1},
{"mhpR" → "mhpABCDE", 1}, {"mlc" → "malT", 2}, {"mlc" → "manXYZ", 2},
{"mlc" → "ptsG", 2}, {"mlc" → "ptsHI_crr", 2}, {"modE" → "modABC", 2},
{"nac" → "gdhA", 2}, {"nac" → "putAP", 1}, {"nadR" → "nadB", 2},
{"nadR" → "pncB", 2}, {"nagBACD" → "glmUS", 3}, {"nagBACD" → "manXYZ", 2},
{"nagBACD" → "nagE", 2}, {"narL" → "adhE", 2}, {"narL" → "caif", 2},
{"narL" → "dcuB_fumB", 2}, {"narL" → "dmsABC", 2}, {"narL" → "fdnGHI", 1},
{"narL" → "focA_pflB", 2}, {"narL" → "frdABCD", 2}, {"narL" → "narGHJI", 1},
{"narL" → "narK", 1}, {"narL" → "nirBDC_cysG", 1}, {"narL" → "nrfABCDEFG", 1},
{"narL" → "nuoABCDEFGHIJKLMNOP", 1}, {"narL" → "torCAD", 2}, {"nhaR" → "nhaA", 1},
{"nlpD_rpoS" → "acs", 1}, {"nlpD_rpoS" → "adhE", 1}, {"nlpD_rpoS" → "aldB", 1},
{"nlpD_rpoS" → "alkA", 1}, {"nlpD_rpoS" → "appY", 1}, {"nlpD_rpoS" → "cpXAR", 1},
{"nlpD_rpoS" → "dps", 1}, {"nlpD_rpoS" → "ftsQAZ", 1}, {"nlpD_rpoS" → "katG", 1},
{"nlpD_rpoS" → "narZYWV", 1}, {"nlpD_rpoS" → "nhaA", 1},
{"nlpD_rpoS" → "osmC", 1}, {"nlpD_rpoS" → "osmY", 1}, {"nlpD_rpoS" → "proP", 1},
{"ompR_envZ" → "csgBA", 1}, {"ompR_envZ" → "csgDEFG", 1},
{"ompR_envZ" → "fadL", 2}, {"ompR_envZ" → "flhDC", 2},
{"ompR_envZ" → "ompC", 1}, {"ompR_envZ" → "ompF", 3}, {"oxyR" → "ahpCF", 1},
{"oxyR" → "dps", 1}, {"oxyR" → "gorA", 1}, {"oxyR" → "katG", 1},
{"phoBR" → "phnCDE_f73_phnFGHIJKLMNOP", 1}, {"phoBR" → "phoA", 1},
{"phoBR" → "phoE", 1}, {"phoBR" → "pstSCAB_phoU", 1}, {"pspF" → "pspABCDE", 1},
{"purR" → "codBA", 2}, {"purR" → "cvpA_purF_ubiX", 2},
{"purR" → "gcvTHP", 2}, {"purR" → "glnB", 2}, {"purR" → "glyA", 2},
{"purR" → "guaBA", 2}, {"purR" → "prsA", 2}, {"purR" → "purC", 2},
{"purR" → "pureK", 2}, {"purR" → "purHD", 2}, {"purR" → "purL", 2},
{"purR" → "purMN", 2}, {"purR" → "pyrC", 2}, {"purR" → "pyrD", 2},
{"purR" → "speA", 2}, {"purR" → "ycfC_purB", 2}, {"rbsR" → "rbsDABC", 2},
{"rcsA" → "ftsQAZ", 1}, {"rcsA" → "wza_wzb_b2060_wcaA_wcaB", 1},
{"rhaSR" → "rhaBAD", 1}, {"rhaSR" → "rhaT", 1}, {"rob" → "aslB", 1},
{"rob" → "fumC", 1}, {"rob" → "galETKM", 2}, {"rob" → "inaA", 1},
{"rob" → "marRAB", 1}, {"rob" → "mdlA", 1}, {"rob" → "nfo", 1},
{"rob" → "soda", 1}, {"rob" → "ybaO", 1}, {"rob" → "ybis", 1},
{"rob" → "yfhd", 1}, {"rob" → "zwf", 1}, {"rpiR_alsBACEK" → "rpiB", 2},
{"rpoE_rseABC" → "cutC", 1}, {"rpoE_rseABC" → "dapA_nlpB_purA", 1},
{"rpoE_rseABC" → "ecfABC", 1}, {"rpoE_rseABC" → "ecfD", 1},
{"rpoE_rseABC" → "ecfF", 1}, {"rpoE_rseABC" → "ecfG", 1},
{"rpoE_rseABC" → "ecfH", 1}, {"rpoE_rseABC" → "ecfI", 1},
{"rpoE_rseABC" → "ecfJ", 1}, {"rpoE_rseABC" → "ecfK", 1},
{"rpoE_rseABC" → "ecfLM", 1}, {"rpoE_rseABC" → "fkpA", 1},
{"rpoE_rseABC" → "htrA", 1}, {"rpoE_rseABC" → "ksgA_epaG_epaH", 1},
{"rpoE_rseABC" → "lpxDA_fabZ", 1}, {"rpoE_rseABC" → "mdoGH", 1},
{"rpoE_rseABC" → "nlpB_purA", 1}, {"rpoE_rseABC" → "osta_surA_pdxA", 1},
{"rpoE_rseABC" → "rfaDFCL", 1}, {"rpoE_rseABC" → "rpoD", 1},
{"rpoE_rseABC" → "rpoH", 1}, {"rpoE_rseABC" → "skp_lpxDA_fabZ", 1},
{"rpoE_rseABC" → "upps_cdsA_ecfE", 1}, {"rpoE_rseABC" → "xprB_dsbC_recJ", 1},
{"rpoH" → "clpP", 1}, {"rpoH" → "dnaKJ", 1}, {"rpoH" → "grpE", 1},
{"rpoH" → "hflB", 1}, {"rpoH" → "htpG", 1}, {"rpoH" → "htpY", 1},
{"rpoH" → "ibpAB", 1}, {"rpoH" → "lon", 1}, {"rpoH" → "mopA", 1},
{"rpoH" → "mopB", 1}, {"rpoH" → "atoC", 1}, {"rpoH" → "dctA", 1},
{"rpoH" → "fdhF", 1}, {"rpoH" → "fhla", 1}, {"rpoH" → "glnALG", 1},
{"rpoH" → "glnHPQ", 1}, {"rpoH" → "hycABCDEFGH", 1}, {"rpoH" → "hypA", 1},
{"rpoH" → "nac", 1}, {"rpoH" → "nycA", 1}, {"rpoH" → "pspABCDE", 1},
{"rpoH" → "rtcR", 1}, {"rpoH" → "zraP", 1}, {"rtcR" → "rtcAB", 2},
{"soxR" → "soxS", 1}, {"soxS" → "acnA", 1}, {"soxS" → "fpr", 1},
{"soxS" → "fumC", 1}, {"soxS" → "nfo", 1}, {"soxS" → "soda", 1},
{"soxS" → "zwf", 1}, {"tdcAR" → "tdcABCDEFG", 1}, {"torR" → "torCAD", 1},
{"treR" → "treBC", 2}, {"trpR" → "aroH", 2}, {"trpR" → "aroL_yaiA_aroM", 2},

```

```

{"trpR" → "mtr", 2}, {"trpR" → "trpLEDcBA", 2}, {"tyrR" → "aroF_tyrA", 2},
{"tyrR" → "aroG", 2}, {"tyrR" → "aroL_yaiA_aroM", 2}, {"tyrR" → "aroP", 2},
{"tyrR" → "mtr", 1}, {"tyrR" → "tyrB", 2}, {"tyrR" → "tyrP", 3},
{"uhpA" → "uhpT", 1}, {"uidR" → "uidRABC", 2}, {"uxuABR" → "uidRABC", 2},
{"xapR" → "xapAB", 1}, {"xylFGHR" → "xylAB", 1}, {"yhdG_fis" → "adhE", 1},
{"yhdG_fis" → "alaWX", 1}, {"yhdG_fis" → "aldB", 3}, {"yhdG_fis" → "argU", 1},
{"yhdG_fis" → "argW", 1}, {"yhdG_fis" → "argX_hisR_leuT_prom", 1},
{"yhdG_fis" → "aspV", 1}, {"yhdG_fis" → "leuQPv", 1},
{"yhdG_fis" → "leuX", 1}, {"yhdG_fis" → "lysT_valT_lysW", 1},
{"yhdG_fis" → "metT_leuW_glnUW_metU_glnVX", 1},
{"yhdG_fis" → "metY_yhbC_nusA_infB", 1}, {"yhdG_fis" → "nrdAB", 1},
{"yhdG_fis" → "pdhR_aceEF_lpdA", 1}, {"yhdG_fis" → "pheU", 1},
{"yhdG_fis" → "pheV", 1}, {"yhdG_fis" → "proK", 1}, {"yhdG_fis" → "proL", 1},
{"yhdG_fis" → "proP", 1}, {"yhdG_fis" → "sdhCDAB_b0725_sucABCD", 1},
{"yhdG_fis" → "serT", 1}, {"yhdG_fis" → "serX", 1},
{"yhdG_fis" → "thru_tyrU_glyT_thrT", 1}, {"yhdG_fis" → "thrW", 1},
{"yhdG_fis" → "tyrTV", 1}, {"yhdG_fis" → "valUXY_lysV", 1},
{"yiaJ" → "yiaKLMNOPQRS", 2}, {"yjbK" → "znuABC", 2}, {"yjdHG" → "dctA", 1},
{"yjdHG" → "dcuB_fumB", 1}, {"yjdHG" → "frdABCD", 1}, {"zntR" → "zntA", 1}];

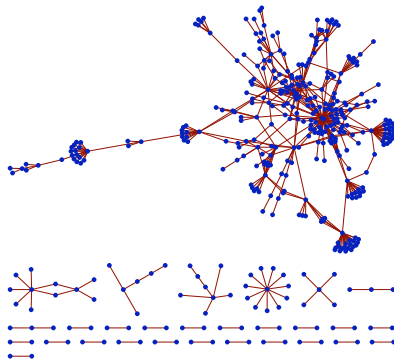
```

Drawing the Network

The network consists of many components. Mouse over vertices to see the labels.

```
In[87]:= GraphPlot[g, EdgeLabeling → Automatic, VertexLabeling → Tooltip]
```

Out[87]=

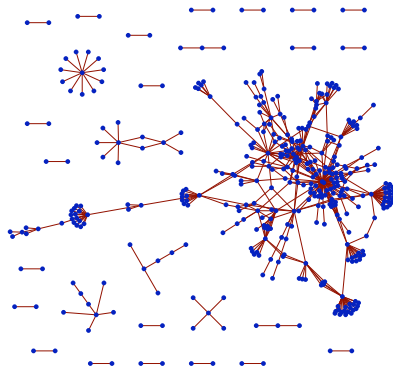


Use a different component packing method.

```
In[88]:= GraphPlot[g, EdgeLabeling → Automatic,  

VertexLabeling → Tooltip, PackingMethod → "ClosestPackingCenter"]
```

Out[88]=



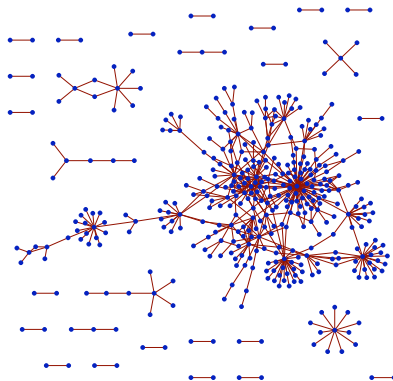
This spreads out the vertices.

```
In[89]:= GraphPlot[g, Method → {"SpringElectricalEmbedding", "RepulsiveForcePower" → -2},  

EdgeLabeling → Automatic, VertexLabeling → Tooltip,  

PackingMethod → "ClosestPackingCenter"]
```

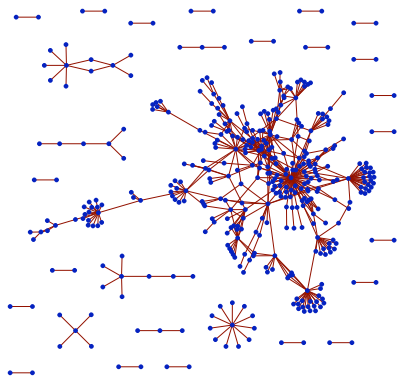
Out[89]=



An alternative way of spreading out the vertices.

```
In[90]:= GraphPlot[g, Method -> {"SpringElectricalEmbedding", "InferentialDistance" -> 6},
EdgeLabeling -> Automatic, VertexLabeling -> Tooltip,
PackingMethod -> "ClosestPackingCenter"]
```

Out[90]=



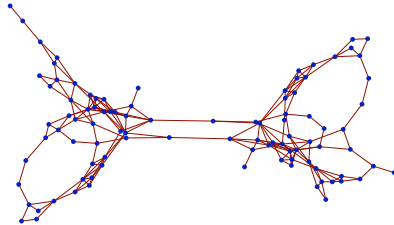
Protein: An Oxidoreductase

This plots an oxidoreductase protein [3] using the data from [4].

```
In[91]:= GraphPlot[{{1 -> 2, 1}, {1 -> 9, 2}, {1 -> 17, 2}, {2 -> 3, 1}, {2 -> 9, 2}, {3 -> 49, 2},
{4 -> 5, 2}, {4 -> 42, 2}, {5 -> 6, 2}, {5 -> 9, 2}, {5 -> 17, 2}, {5 -> 42, 1}, {6 -> 7, 1},
{6 -> 8, 2}, {6 -> 17, 2}, {6 -> 28, 2}, {6 -> 42, 2}, {7 -> 8, 2}, {7 -> 16, 2}, {7 -> 28, 2},
{7 -> 44, 2}, {8 -> 10, 2}, {8 -> 11, 2}, {8 -> 14, 2}, {8 -> 16, 2}, {8 -> 18, 2},
{8 -> 19, 2}, {8 -> 28, 2}, {9 -> 17, 2}, {10 -> 11, 1}, {10 -> 12, 2}, {10 -> 13, 2},
{10 -> 14, 2}, {10 -> 17, 2}, {10 -> 18, 2}, {11 -> 12, 2}, {11 -> 18, 2}, {12 -> 13, 2},
{12 -> 14, 2}, {12 -> 18, 2}, {13 -> 14, 2}, {13 -> 15, 2}, {13 -> 16, 2}, {13 -> 18, 2},
{13 -> 32, 2}, {14 -> 15, 1}, {14 -> 16, 2}, {14 -> 18, 2}, {14 -> 32, 1}, {14 -> 36, 2},
{15 -> 16, 2}, {15 -> 32, 2}, {15 -> 36, 2}, {15 -> 37, 2}, {16 -> 17, 2}, {16 -> 18, 2},
{16 -> 32, 2}, {16 -> 33, 2}, {16 -> 37, 2}, {18 -> 19, 2}, {19 -> 20, 2}, {19 -> 43, 2},
{20 -> 21, 2}, {20 -> 43, 2}, {21 -> 22, 2}, {22 -> 27, 2}, {22 -> 28, 2}, {22 -> 29, 2},
{23 -> 24, 2}, {23 -> 25, 2}, {23 -> 26, 2}, {23 -> 27, 2}, {23 -> 30, 2}, {23 -> 31, 2},
{23 -> 39, 2}, {24 -> 25, 2}, {24 -> 31, 2}, {24 -> 39, 2}, {25 -> 26, 2}, {25 -> 31, 2},
{26 -> 27, 2}, {26 -> 30, 2}, {26 -> 31, 2}, {26 -> 32, 2}, {27 -> 30, 2}, {28 -> 29, 2},
{29 -> 85, 2}, {30 -> 31, 2}, {30 -> 32, 2}, {30 -> 33, 2}, {32 -> 36, 2}, {32 -> 85, 2},
{33 -> 36, 2}, {35 -> 36, 2}, {35 -> 79, 2}, {35 -> 82, 2}, {36 -> 37, 2}, {37 -> 38, 2},
{39 -> 40, 2}, {39 -> 41, 2}, {39 -> 47, 2}, {40 -> 41, 2}, {40 -> 46, 2}, {40 -> 48, 2},
{43 -> 44, 2}, {44 -> 45, 1}, {45 -> 46, 1}, {47 -> 48, 2}, {50 -> 51, 1}, {50 -> 52, 2},
{50 -> 59, 1}, {50 -> 67, 2}, {51 -> 52, 2}, {51 -> 59, 2}, {51 -> 67, 2}, {52 -> 53, 1},
{53 -> 93, 2}, {53 -> 99, 2}, {54 -> 55, 2}, {54 -> 92, 2}, {55 -> 56, 2}, {55 -> 59, 2},
{55 -> 67, 2}, {55 -> 92, 1}, {56 -> 57, 1}, {56 -> 58, 2}, {56 -> 67, 2}, {56 -> 78, 2},
{56 -> 92, 2}, {57 -> 58, 2}, {57 -> 66, 2}, {57 -> 78, 2}, {57 -> 94, 2}, {58 -> 60, 2},
{58 -> 61, 2}, {58 -> 64, 2}, {58 -> 66, 2}, {58 -> 68, 2}, {58 -> 69, 2}, {58 -> 78, 2},
{59 -> 67, 2}, {60 -> 61, 2}, {60 -> 62, 2}, {60 -> 64, 2}, {60 -> 67, 2}, {60 -> 68, 2},
{61 -> 62, 2}, {61 -> 68, 2}, {62 -> 63, 2}, {62 -> 68, 2}, {63 -> 64, 2}, {63 -> 65, 2},
{63 -> 66, 2}, {63 -> 68, 2}, {63 -> 82, 2}, {64 -> 65, 1}, {64 -> 66, 2}, {64 -> 68, 2},
{64 -> 82, 1}, {64 -> 86, 2}, {65 -> 66, 2}, {65 -> 82, 2}, {65 -> 86, 2}, {65 -> 87, 2},
{66 -> 67, 2}, {66 -> 68, 2}, {66 -> 82, 2}, {66 -> 83, 2}, {68 -> 69, 2},
```

```
{69 → 70, 2}, {69 → 93, 2}, {70 → 71, 2}, {71 → 72, 2}, {72 → 77, 2}, {72 → 78, 2},
{72 → 79, 2}, {73 → 74, 2}, {73 → 75, 2}, {73 → 76, 2}, {73 → 77, 2}, {73 → 80, 2},
{73 → 81, 2}, {73 → 89, 2}, {74 → 75, 2}, {74 → 81, 2}, {74 → 89, 2}, {75 → 76, 2},
{75 → 81, 2}, {76 → 77, 2}, {76 → 80, 2}, {76 → 81, 2}, {76 → 82, 2}, {77 → 80, 2},
{78 → 79, 2}, {80 → 81, 2}, {80 → 82, 2}, {80 → 83, 2}, {82 → 86, 2}, {85 → 86, 2},
{86 → 87, 2}, {87 → 88, 2}, {89 → 90, 2}, {89 → 91, 2}, {89 → 97, 2}, {90 → 91, 2},
{90 → 96, 2}, {90 → 98, 2}, {93 → 94, 2}, {94 → 95, 1}, {95 → 96, 1}, {97 → 98, 2}},
EdgeLabeling → Automatic, MultiedgeStyle → False]
```

Out[91]=



Square Dielectric Waveguide

A square sparse matrix can be viewed as an adjacency matrix of a graph; therefore it is often instructive to "draw" the sparse matrix using `GraphPlot`. An example is given below, and the graph drawing of over one thousand matrices can be found at [7].

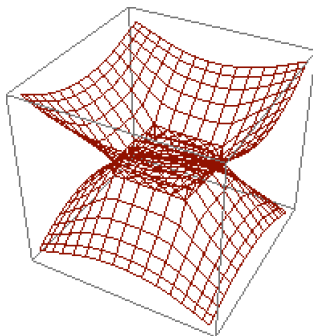
This graph represents a sparse matrix used in electrical engineering [5].

```
In[113]:= g = Import["ftp://math.nist.gov/pub/MatrixMarket2/NEP/dwave/dwa512.mtx.gz"]
```

```
Out[115]= SparseArray[<2480>, {512, 512}]
```

```
In[116]:= GraphPlot3D[g, VertexRenderingFunction → None]
```

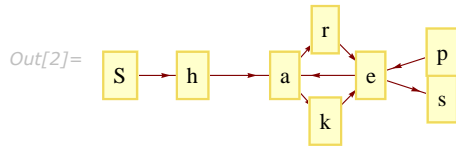
Out[116]=



This generates a graph by linking each letter in a word to all the letters that follow it in the word.

```
In[1]:= WordPlot[w_String] := GraphPlot[(x = Characters[w];
  Thread[Drop[x, -1] → Drop[x, 1]]), VertexLabeling → True, DirectedEdges → True];
```

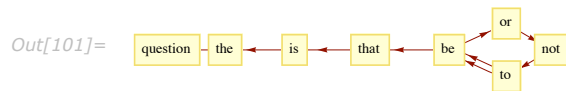
```
In[2]:= WordPlot["Shakespeare"]
```



This generates a graph by linking words in a text with subsequent words.

```
In[100]:= TextPlot[w_String] :=
  GraphPlot[(x = Map[ToLowerCase, StringCases[w, WordCharacter ..]];
  g = Thread[Drop[x, -1] → Drop[x, 1]];
  g), DirectedEdges → True, VertexLabeling → True];
```

```
In[101]:= TextPlot["to be or not to be, that is the question."]
```

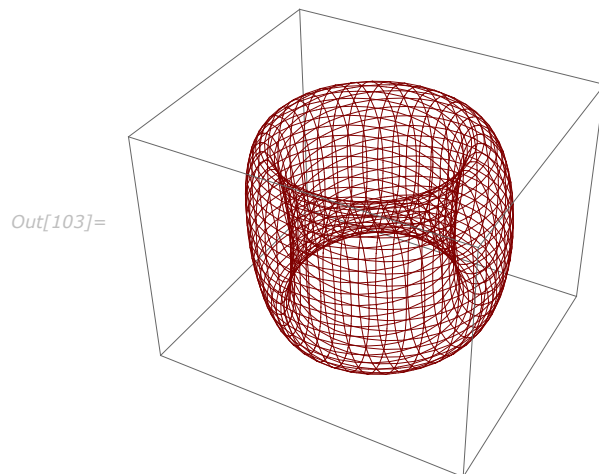


Torus

This defines a torus and plots it in 3D.

```
In[102]:= Torus[m_, n_] :=
  SparseArray[
    Flatten[Table[{{i * n + j + 1, Mod[i + 1, m] * n + j + 1} ->
      1, {i * n + j + 1, Mod[i - 1, m] * n + j + 1} ->
      1, {i * n + j + 1, i * n + Mod[j + 1, n] + 1} ->
      1, {i * n + j + 1, i * n + Mod[j - 1, n] + 1} -> 1}, {i, 0,
      m - 1}, {j, 0, n - 1}], {m * n, m * n}];
```

```
In[103]:= GraphPlot3D[Torus[40, 40], VertexRenderingFunction → None]
```



References

- [1] Milo, R., S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. "Network Motifs: Simple Building Blocks of Complex Networks." *Science* 298, no. 5594 (2002): 824-827.
- [2] Alon, U. "Collection of Complex Networks." *Uri Alon Homepage* 2007.
<http://www.weizmann.ac.il/mcb/UriAlon/groupNetworksData.html>
- [3] Milo, R., S. Itzkovitz, N. Kashtan, et al. "Superfamilies of Designed and Evolved Networks." *Science* 303, no. 5663 (2004): 1538-1542.
- [4] Alon, U. "1AORInter." *network Motifs* 2007.
http://www.weizmann.ac.il/mcb/UriAlon/Papers/networkMotifs/1AORInter_st.txt
- [5] National Institute of Standards and Technology. "DWA512: Square Dielectric Waveguide." *Matrix Market* 2007. <http://math.nist.gov/MatrixMarket/data/NEP/dwave/dwa512.html>
- [6] Freivalds, K., U. Dogrusoz, and P. Kikusts, "Disconnected Graph Layout and the Polyomino Packing Approach." *Lecture Notes in Computer Science: Revised Papers from the 9th International Symposium on Graph Drawing* 2265 (2001): 378-391.
- [7] Hu, Y. F. "Graph Drawing of Square Matrices from University of Florida Sparse Matrix Collection." (2007). <http://members.wolfram.com/~yifanhu/UFMatrixGraphPlot>
- [8] Hu, Y. F. "Efficient, High-Quality Force-Directed Graph Drawing." *The Mathematica Journal* 10, no. 1 (2006): 37-71.

Hierarchical Drawing of Directed Graphs

`LayeredGraphPlot` attempts to draw the vertices of a graph in a series of layers, placing dominant vertices at the top, and vertices lower in the hierarchy progressively further down.

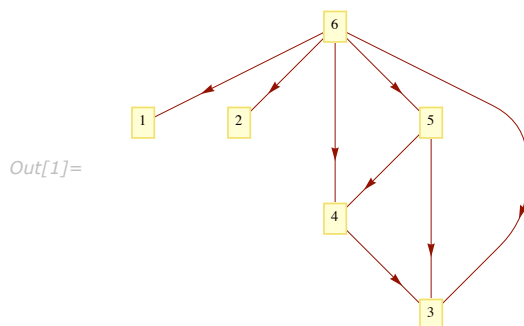
<code>LayeredGraphPlot</code> [$\{v_{i1} \rightarrow v_{j1}, v_{i2} \rightarrow v_{j2}, \dots\}$]	generate a layered plot of the graph in which vertex v_{ik} is connected to vertex v_{jk}
<code>LayeredGraphPlot</code> [$\{ \{v_{i1} \rightarrow v_{j1}, lbl_1\}, \dots \}$]	associate labels lbl_k with edges in the graph
<code>LayeredGraphPlot</code> [g, pos]	place the dominant vertices in the plot at position pos
<code>LayeredGraphPlot</code> [m]	generate a layered plot of the graph represented by the adjacency matrix m

Hierarchical graph drawing.

`LayeredGraphPlot` draws a graph so that the edges point predominantly downward. The second argument of `LayeredGraphPlot` specifies the position of the root. Possible values for this argument are `Right`, `Left`, `Top`, and `Bottom`.

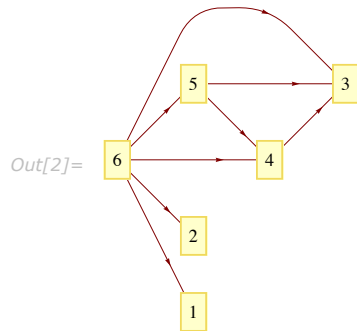
This plots a directed graph.

```
In[1]:= LayeredGraphPlot[
  {4 → 3, 5 → 3, 5 → 4, 6 → 1, 6 → 2, 6 → 4, 6 → 5, 6 → 3}, VertexLabeling -> True]
```



This is the same graph, with edges pointing from left to right.

```
In[2]:= LayeredGraphPlot[{4 → 3, 5 → 3, 5 → 4, 6 → 1, 6 → 2, 6 → 4, 6 → 5, 6 → 3},
  Left, VertexLabeling -> True]
```



`LayeredGraphPlot` may produce slightly different output on different platforms, due to floating-point differences.

Options for LayeredGraphPlot

In addition to options for `Graphics`, the following options are accepted for `LayeredGraphPlot`.

<i>option name</i>	<i>default value</i>	
<code>DataRange</code>	<code>Automatic</code>	the range of vertex coordinates to generate
<code>DirectedEdges</code>	<code>True</code>	whether to show edges as directed arrows
<code>EdgeLabeling</code>	<code>True</code>	whether to include labels given for edges
<code>EdgeRenderingFunction</code>	<code>Automatic</code>	function to give explicit graphics for edges
<code>MultiedgeStyle</code>	<code>Automatic</code>	how to draw multiple edges between vertices
<code>PackingMethod</code>	<code>Automatic</code>	method to use for packing components
<code>PlotRangePadding</code>	<code>Automatic</code>	how much padding to put around the plot
<code>PlotStyle</code>	<code>Automatic</code>	style in which objects are drawn
<code>SelfLoopStyle</code>	<code>Automatic</code>	how to draw edges linking a vertex to itself
<code>VertexCoordinateRules</code>	<code>Automatic</code>	rules for explicit vertex coordinates
<code>VertexLabeling</code>	<code>Automatic</code>	whether to show vertex names as labels
<code>VertexRenderingFunction</code>	<code>Automatic</code>	function to give explicit graphics for vertices

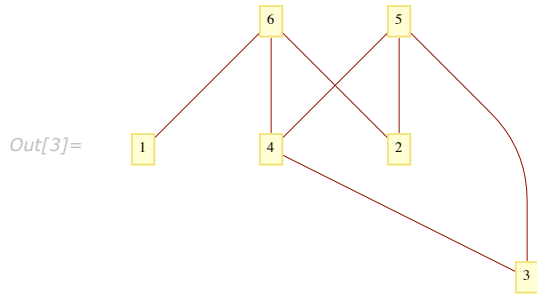
Options for `LayeredGraphPlot`.

DirectedEdges

The option `DirectedEdges` specifies whether to draw edges as arrows. Possible values for this option are `True` or `False`. The default value for this option is `True`.

This shows a graph with edges represented by lines instead of arrows.

```
In[3]:= LayeredGraphPlot[{4 → 3, 5 → 2, 5 → 3, 5 → 4, 6 → 1, 6 → 2, 6 → 4},
  DirectedEdges → False, VertexLabeling → True]
```

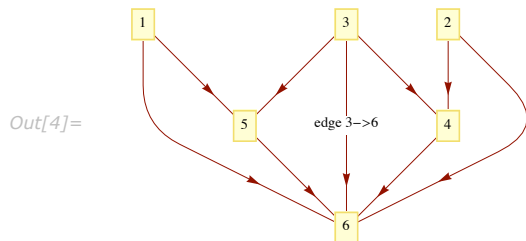


EdgeLabeling

The option `EdgeLabeling` specifies whether and how to display labels given for the edges. Possible values for this option are `True`, `False`, or `Automatic`. The default value for this option is `True`, which displays the supplied edge labels on the graph. With `EdgeLabeling -> Automatic`, the labels are shown as tooltips.

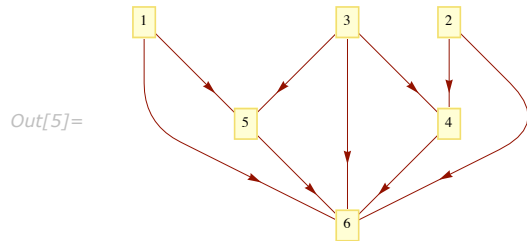
This displays the specified edge label.

```
In[4]:= LayeredGraphPlot[{1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4,
  3 → 5, {3 → 6, "edge 3->6"}, 4 → 6, 5 → 6}, VertexLabeling → True]
```



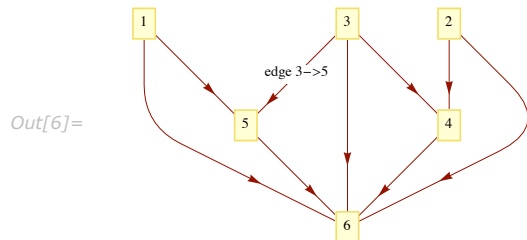
This displays the labels as tooltips. Place the cursor over the edge between vertices 3 and 6 to see the tooltip.

```
In[5]:= LayeredGraphPlot[
  {1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4, 3 → 5, {3 → 6, "edge 3->6"}, 4 → 6, 5 → 6},
  EdgeLabeling → Automatic, VertexLabeling → True]
```



Alternatively, use `Tooltip[vi -> vj, lbl]` to specify a tooltip for an edge. Place the cursor over the edge between vertices 3 and 6, as well as the edge label on the edge between vertices 3 and 5, to see the tooltips.

```
In[6]:= LayeredGraphPlot[
  {1 → 5, 1 → 6, 2 → 4, 2 → 6, 3 → 4, {3 → 5, Tooltip["edge 3->5", "3->5"]},
  Tooltip[3 → 6, "3->6"], 4 → 6, 5 → 6}, VertexLabeling → True]
```

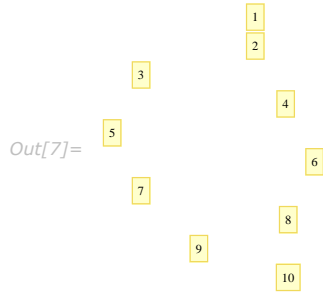


EdgeRenderingFunction

The option `EdgeRenderingFunction` specifies graphical representation of the graph edges. Possible values for this option are `Automatic`, `None`, or a function that gives a proper combination of graphics primitives and directives. With the default setting of `Automatic`, a dark red line is drawn for each edge. With `EdgeRenderingFunction -> None`, edges are not drawn.

This draws vertices only.

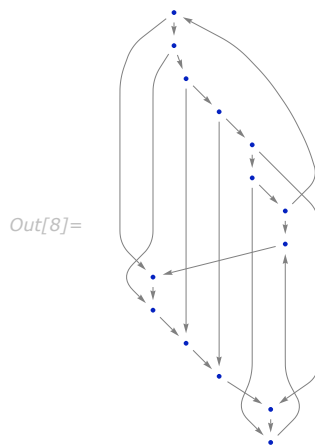
```
In[7]:= LayeredGraphPlot[Table[1, {10}], {10}],
EdgeRenderingFunction -> None, VertexLabeling -> True]
```



With `EdgeRenderingFunction -> g`, each edge is rendered with the graphics primitives and directives given by the function `g`. It can take three or more arguments in the form `g[{ri, ..., rj}, {vi, vj}, lblij, ...]`, where `ri`, `rj` are the coordinates of the beginning and ending points of the edge, `vi`, `vj` are the beginning and ending vertices, and `lblij` is any label specified for the edge or `None`. Explicit settings for `EdgeRenderingFunction -> g` override settings for `EdgeLabeling` and `DirectedEdges`.

This plots edges as gray arrows with ends set back from vertices by a distance of 0.3 (in the graph's coordinate system).

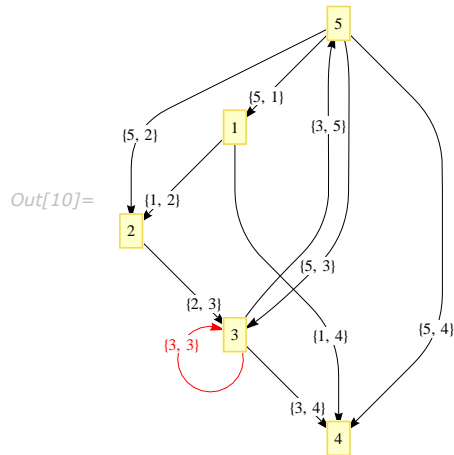
```
In[8]:= LayeredGraphPlot[
{1 -> 2, 2 -> 3, 3 -> 4, 4 -> 5, 5 -> 6, 6 -> 7, 7 -> 1, 11 -> 12, 12 -> 13, 13 -> 14, 14 -> 15,
15 -> 16, 16 -> 17, 17 -> 11, 1 -> 11, 2 -> 12, 3 -> 13, 4 -> 14, 5 -> 15, 6 -> 16, 7 -> 17},
EdgeRenderingFunction -> ({GrayLevel[0.5], Arrow[#1, 0.3]} &)]
```



This displays edges and self-loops with black and red arrows, respectively. The function `LineScaledCoordinate` from the Graph Utilities Package adds text at 70% along arrows.

```
In[9]:= << GraphUtilities`
```

```
In[10]:= LayeredGraphPlot[{1 → 2, 2 → 3, 3 → 4, 5 → 1, 5 → 2, 5 → 3, 5 → 4, 1 → 4, 3 → 5, 3 → 3},
  EdgeRenderingFunction →
  ({If[First[#2] === Last[#2], Red, Black], Arrow[#1, .1], Text[#2,
    LineScaledCoordinate[#1, .7], Background → White]} &), VertexLabeling → True]
```

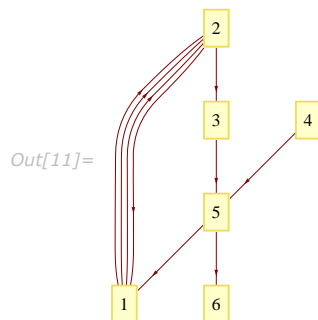


MultiedgeStyle

The option `MultiedgeStyle` specifies whether to draw multiple edges between two vertices. Possible values for `MultiedgeStyle` are `Automatic` (the default), `True`, `False`, or a positive real number. With the default setting `MultiedgeStyle -> Automatic`, multiple edges are shown for a graph specified by a list of rules, but not shown if the graph is specified by an adjacency matrix. With `MultiedgeStyle -> δ` , the multiedges are spread out to a scaled distance of δ .

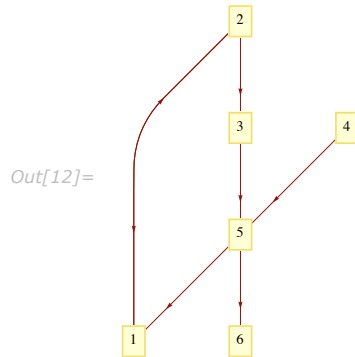
By default, multiple edges are shown if a graph is given as a list of rules.

```
In[11]:= LayeredGraphPlot[
  {1 → 2, 2 → 1, 1 → 2, 1 → 2, 2 → 3, 3 → 5, 4 → 5, 5 → 6, 5 → 1}, VertexLabeling → True]
```



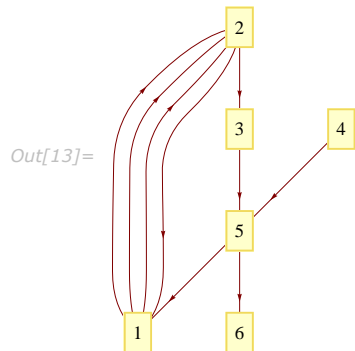
But multiple edges are not shown for graphs specified by an adjacency matrix.

```
In[12]:= LayeredGraphPlot[ $\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ , VertexLabeling → True]
```



This spreads multiple edges by the specified amount.

```
In[13]:= LayeredGraphPlot[{1 → 2, 2 → 1, 1 → 2, 1 → 2, 2 → 3, 3 → 5, 4 → 5, 5 → 6, 5 → 1},
MultiedgeStyle → 0.25, VertexLabeling → True]
```

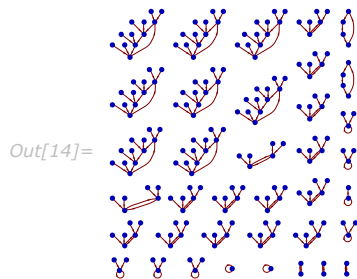


PackingMethod

The option `PackingMethod` specifies the method used for packing disconnected components. Possible values for the option are `Automatic` (the default), `"ClosestPacking"`, `"ClosestPackingCenter"`, `"Layered"`, `"LayeredLeft"`, `"LayeredTop"`, and `"NestedGrid"`. With `PackingMethod -> "ClosestPacking"`, components are packed as close together as possible using a polyomino method [6], starting from the top left. With `PackingMethod -> "ClosestPackingCenter"`, components are packed starting from the center. With `PackingMethod -> "Layered"`, components are packed in layers starting from the top left. With `PackingMethod -> "LayeredLeft"` or `PackingMethod -> "LayeredTop"`, components are packed in layers starting from the top/left respectively. With `PackingMethod -> "NestedGrid"`, components are arranged in a nested grid. The typical effective default setting is `PackingMethod -> "Layered"`, and the packing starts with components of the largest bounding box area.

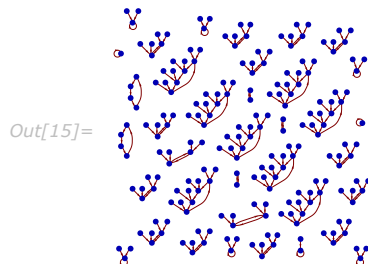
This shows the packing of disconnected components by the default method.

```
In[14]:= LayeredGraphPlot[Table[i -> Mod[i^3, 221], {i, 0, 221}]]
```



This shows the packing of disconnected components using the `"ClosestPackingCenter"` method.

```
In[15]:= LayeredGraphPlot[Table[i -> Mod[i^3, 221], {i, 0, 221}],  
PackingMethod -> "ClosestPackingCenter"]
```



PlotRangePadding

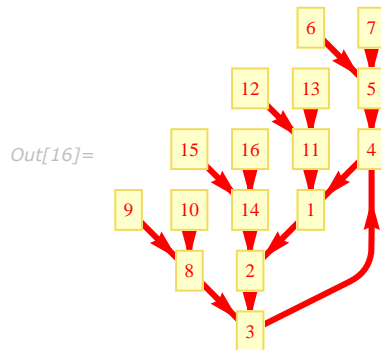
`PlotRangePadding` is a common option for graphics functions inherited by `LayeredGraphPlot`.

PlotStyle

`PlotStyle` is a common option for graphics functions inherited by `LayeredGraphPlot`. The option `PlotStyle` specifies the style in which objects are drawn.

Draw edges with thicker arrows, and both edges and vertices' labels in red.

```
In[16]:= LayeredGraphPlot[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 6 → 5, 7 → 5, 5 → 4, 9 → 8, 10 → 8, 8 → 3,
  12 → 11, 13 → 11, 11 → 1, 15 → 14, 16 → 14, 14 → 2}, VertexLabeling → True,
  PlotStyle → {Red, Arrowheads[{{0.1, 0.8}}], Thickness[0.02]}
```

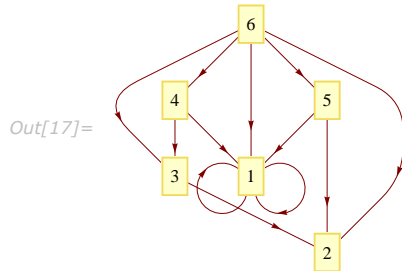


SelfLoopStyle

The option `SelfLoopStyle` specifies whether and how to draw loops for vertices that are linked to themselves. Possible values of the option are `Automatic` (the default), `True`, `False`, or a positive real number. With `SelfLoopStyle -> Automatic`, self-loops are shown if the graph is specified by a list of rules, but not by an adjacency matrix. With `SelfLoopStyle -> δ` , the self-loops are drawn with a diameter of δ (relative to the average edge length).

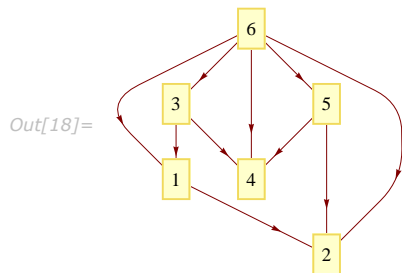
By default, self-loops are displayed for a graph specified by a list of rules.

```
In[17]:= LayeredGraphPlot[{3 → 2, 4 → 1, 4 → 3, 5 → 1, 5 → 2, 6 → 1,
6 → 2, 6 → 3, 6 → 4, 6 → 5, 1 → 1, 1 → 1}, VertexLabeling → True]
```



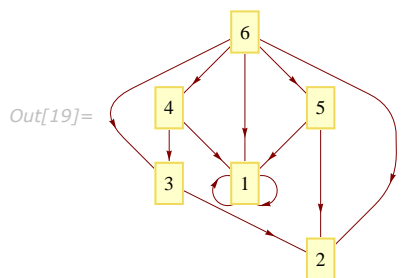
Self-loops are not shown if the graph is specified by an adjacency matrix.

```
In[18]:= LayeredGraphPlot[ $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$ , VertexLabeling → True]
```



This shows self-loops with diameter equal to 0.3 times the average length of the edges.

```
In[19]:= LayeredGraphPlot[
{3 → 2, 4 → 1, 4 → 3, 5 → 1, 5 → 2, 6 → 1, 6 → 2, 6 → 3, 6 → 4, 6 → 5, 1 → 1, 1 → 1},
VertexLabeling → True, SelfLoopStyle → 0.3]
```



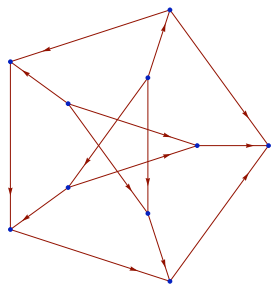
VertexCoordinateRules

The option `VertexCoordinateRules` specifies the coordinates of the vertices. Possible values are `None`, or a list of coordinates. Coordinates specified by a list of rules are not currently supported by `LayeredGraphPlot`.

This draws the Petersen graph using known coordinates.

```
In[20]:= LayeredGraphPlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7,
7 → 8, 8 → 9, 9 → 10, 6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10},
VertexCoordinateRules → {{0.30901699437494745`, 0.9510565162951535`},
{-0.8090169943749476`, -0.587785252292473`}, {0.30901699437494723`,
-0.9510565162951536`}, {-0.8090169943749473`, 0.5877852522924732`},
{1., 0}, {0.6180339887498949`, 1.902113032590307`}, {-1.6180339887498947`,
1.1755705045849465`}, {-1.6180339887498951`, -1.175570504584946`},
{0.6180339887498945`, -1.9021130325903073`}, {2., 0}}]
```

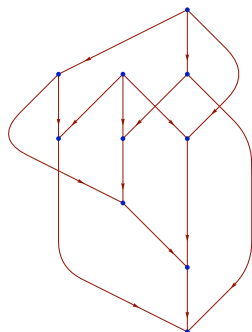
Out[20]=



This draws with the default method.

```
In[21]:= LayeredGraphPlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7,
7 → 8, 8 → 9, 9 → 10, 6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10}]
```

Out[21]=

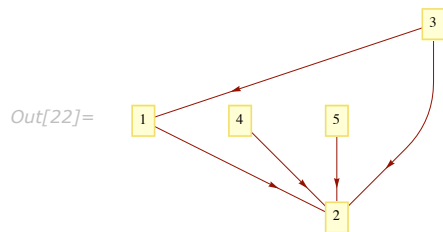


VertexLabeling

The option `VertexLabeling` specifies whether to show vertex names as labels. Possible values for this option are `True`, `False`, `Automatic` (the default) and `Tooltip`. `VertexLabeling -> True` shows the labels. For graphs specified by an adjacency matrix, vertex labels are taken to be successive integers $1, 2, \dots, n$, where n is the size of the matrix. For graphs specified by a list of rules, labels are the expressions used in the rules. `VertexLabeling -> False` displays each vertex as a point. `VertexLabeling -> Tooltip` displays each vertex as a point, but gives its name in a tooltip. `VertexLabeling -> Automatic` displays each vertex as a point, giving its name in a tooltip if the number of vertices is not too large. You can also use `Tooltip[vk, vlbl]` anywhere in the list of rules to specify an alternative tooltip for a vertex v_k .

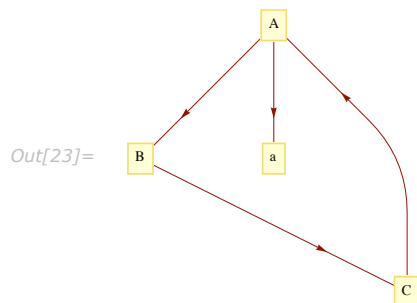
This draws the graph with labels given as indices of the adjacency matrix.

```
In[22]:= LayeredGraphPlot[ $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$ , VertexLabeling -> True]
```



This uses the labels specified in the list of rules.

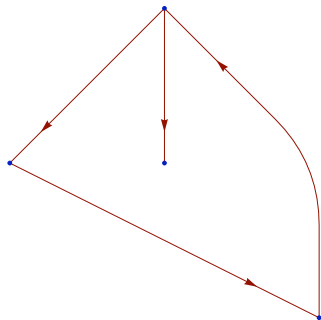
```
In[23]:= LayeredGraphPlot[{"A" -> "B", "A" -> "a", "B" -> "C", "C" -> "A"}, VertexLabeling -> True]
```



This plots vertices as points, and displays vertex names in tooltips. Place the cursor above the vertices to see the labels.

```
In[24]:= LayeredGraphPlot[
  {"A" → "B", "A" → "a", "B" → "C", "C" → "A"}, VertexLabeling → Tooltip]
```

Out[24]=



VertexRenderingFunction

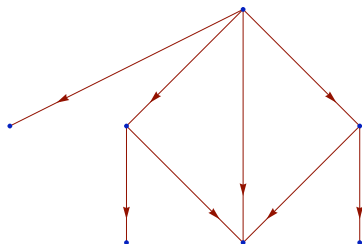
The option `VertexRenderingFunction` specifies graphical representation of the graph edges. Possible values for this option are `Automatic`, `None`, or a function that gives a proper combination of graphics primitives and directives. With the default setting of `Automatic`, vertices are displayed as points, with their names given in tooltips.

By default, vertices are displayed as points and, for small graphs, labeled in tooltips. Point the cursor at a vertex to see the tooltip.

```
In[22]:= g = {5 → 3, 5 → 4, 6 → 2, 6 → 4, 7 → 1, 7 → 4, 7 → 5, 7 → 6};
```

```
In[23]:= LayeredGraphPlot[g]
```

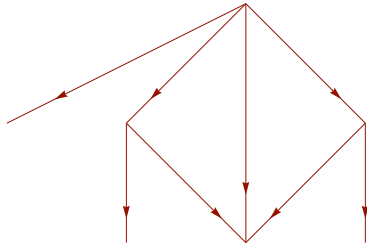
Out[23]=



This draws the same graph, but without the vertices.

```
In[24]:= LayeredGraphPlot[g, VertexRenderingFunction -> None]
```

Out[24]=



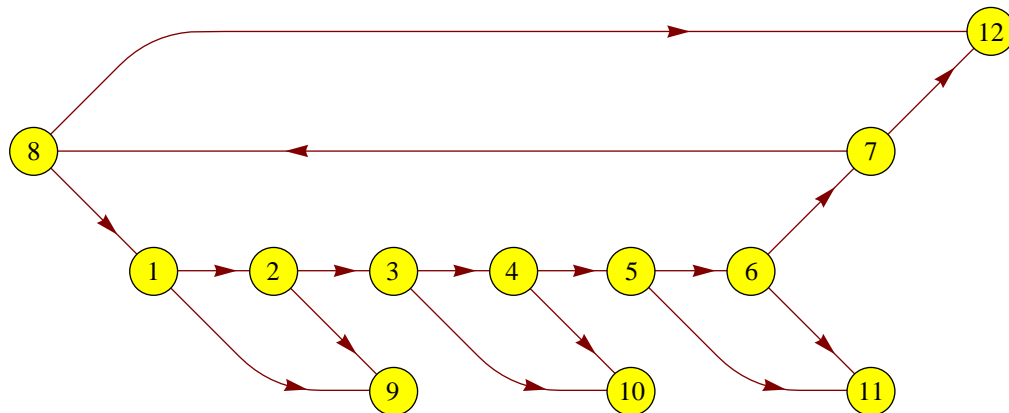
With `VertexRenderingFunction -> g`, each vertex is rendered with the graphics primitives given by $g[r_i, v_i, \dots]$, where r_i is the coordinate of the vertex and v_i is the label of the vertex.

Explicit settings for `VertexRenderingFunction -> g` override settings for `VertexLabeling`.

This shows vertices as yellow disks.

```
In[27]:= LayeredGraphPlot[{1 -> 2, 2 -> 3, 3 -> 4, 4 -> 5, 5 -> 6, 6 -> 7, 7 -> 8, 8 -> 1, 1 -> 9, 2 -> 9,
  3 -> 10, 4 -> 10, 6 -> 11, 5 -> 11, 7 -> 12, 8 -> 12}, Left, VertexRenderingFunction ->
  ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]
```

Out[27]=



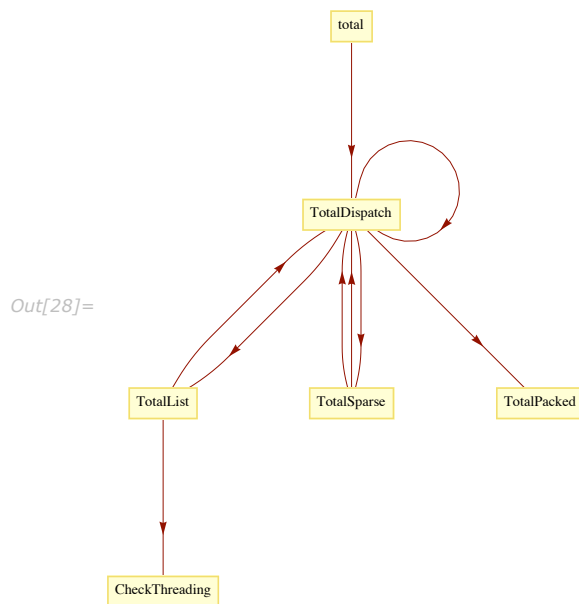
Example Gallery

Flow Chart

LayeredGraphPlot helps visualize flow charts, for example for business, economic, or technical presentations.

This shows a flow chart.

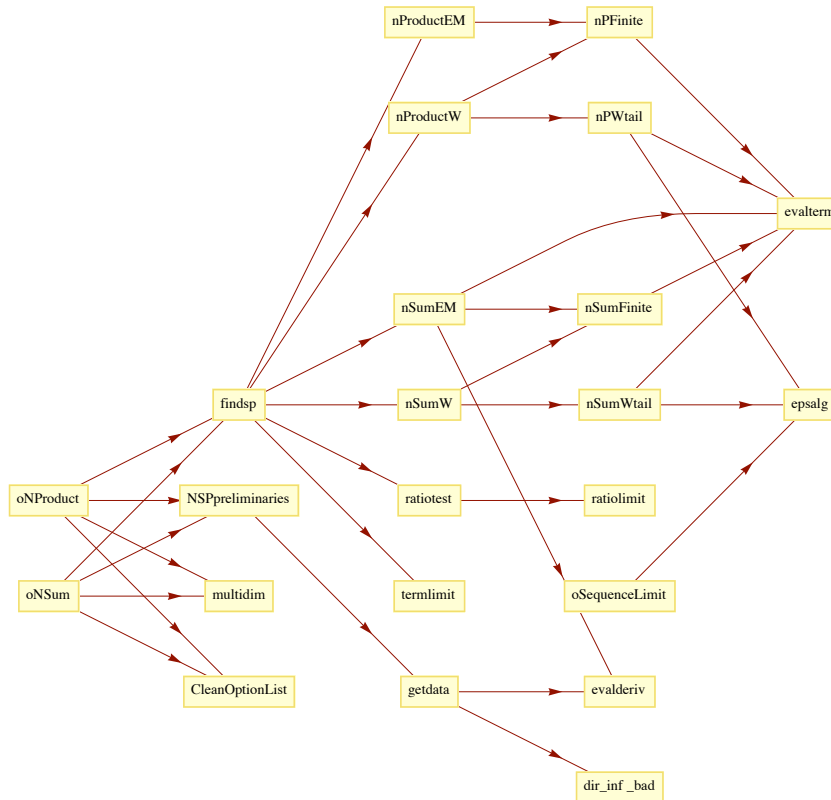
```
In[28]:= LayeredGraphPlot[{"total" → "TotalDispatch", "TotalList" → "CheckThreading",
  "TotalList" → "TotalDispatch", "TotalSparse" → "TotalDispatch",
  "TotalSparse" → "TotalDispatch", "TotalDispatch" → "TotalDispatch",
  "TotalDispatch" → "TotalList", "TotalDispatch" → "TotalPacked",
  "TotalDispatch" → "TotalSparse"}, VertexLabeling → True]
```



This shows a flow chart that flows from left to right.

```
In[29]:= LayeredGraphPlot[
{"ratiotest" -> "ratiolimit", "getdata" -> "dir_inf_bad", "getdata" -> "evalderiv",
"NSPpreliminaries" -> "getdata", "findsp" -> "termlimit", "findsp" -> "ratiotest",
"findsp" -> "nSumW", "findsp" -> "nSumEM", "findsp" -> "nProductEM",
"findsp" -> "nProductW", "nSumW" -> "nSumFinite", "nSumW" -> "nSumWtail",
"nSumEM" -> "evalterm", "nSumEM" -> "nSumFinite", "nSumEM" -> "evalderiv",
"nProductEM" -> "nPFinite", "nProductW" -> "nPFinite", "nProductW" -> "nPWtail",
"oNSum" -> "CleanOptionList", "oNSum" -> "multidim", "oNSum" -> "NSPpreliminaries",
"oNSum" -> "findsp", "nSumFinite" -> "evalterm", "nSumWtail" -> "evalterm",
"nSumWtail" -> "epsalg", "oNProduct" -> "CleanOptionList",
"oNProduct" -> "multidim", "oNProduct" -> "NSPpreliminaries",
"oNProduct" -> "findsp", "nPFinite" -> "evalterm", "nPWtail" -> "evalterm",
"nPWtail" -> "epsalg", "oSequenceLimit" -> "epsalg"}, Left,
VertexLabeling -> True, AspectRatio -> 1, PlotRangePadding -> 0.02]
```

Out[29]=

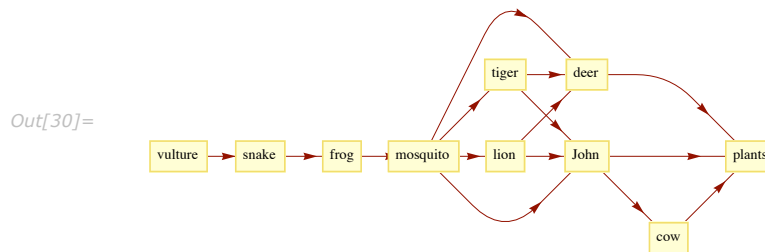


Food Chains

Food chains can be visualized with `LayeredGraphPlot`.

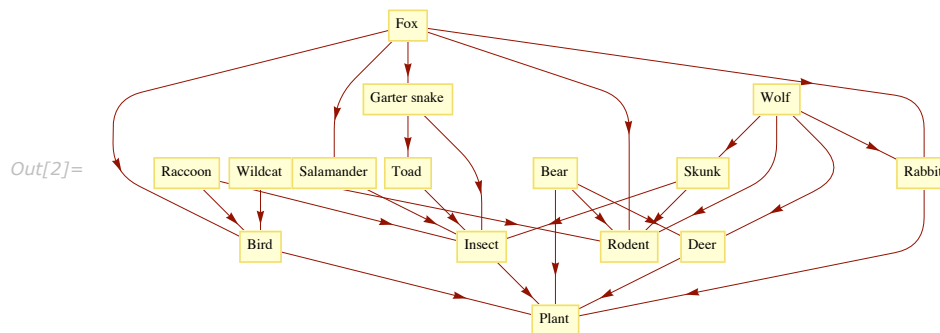
This shows a small food chain.

```
In[30]:= LayeredGraphPlot[{"John" -> "plants",
  "lion" -> "John", "tiger" -> "John",
  "tiger" -> "deer", "lion" -> "deer", "deer" -> "plants",
  "mosquito" -> "lion", "frog" -> "mosquito", "mosquito" -> "tiger",
  "John" -> "cow", "cow" -> "plants", "mosquito" -> "deer",
  "mosquito" -> "John", "snake" -> "frog", "vulture" -> "snake"}, Left,
  VertexLabeling -> True]
```



This shows another food chain.

```
In[2]:= LayeredGraphPlot[{"Raccoon" -> "Bird", "Raccoon" -> "Insect",
  "Wildcat" -> "Bird", "Wildcat" -> "Rodent", "Fox" -> "Bird",
  "Fox" -> "Garter snake", "Fox" -> "Salamander", "Fox" -> "Rabbit",
  "Fox" -> "Rodent", "Wolf" -> "Rabbit", "Wolf" -> "Rodent", "Wolf" -> "Skunk",
  "Wolf" -> "Deer", "Bear" -> "Deer", "Bear" -> "Rodent", "Bear" -> "Plant",
  "Bird" -> "Plant", "Garter snake" -> "Insect", "Garter snake" -> "Toad",
  "Salamander" -> "Insect", "Rabbit" -> "Plant", "Skunk" -> "Rodent",
  "Skunk" -> "Insect", "Deer" -> "Plant", "Toad" -> "Insect", "Insect" -> "Plant"}],
  VertexLabeling -> True, PlotRangePadding -> Automatic]
```

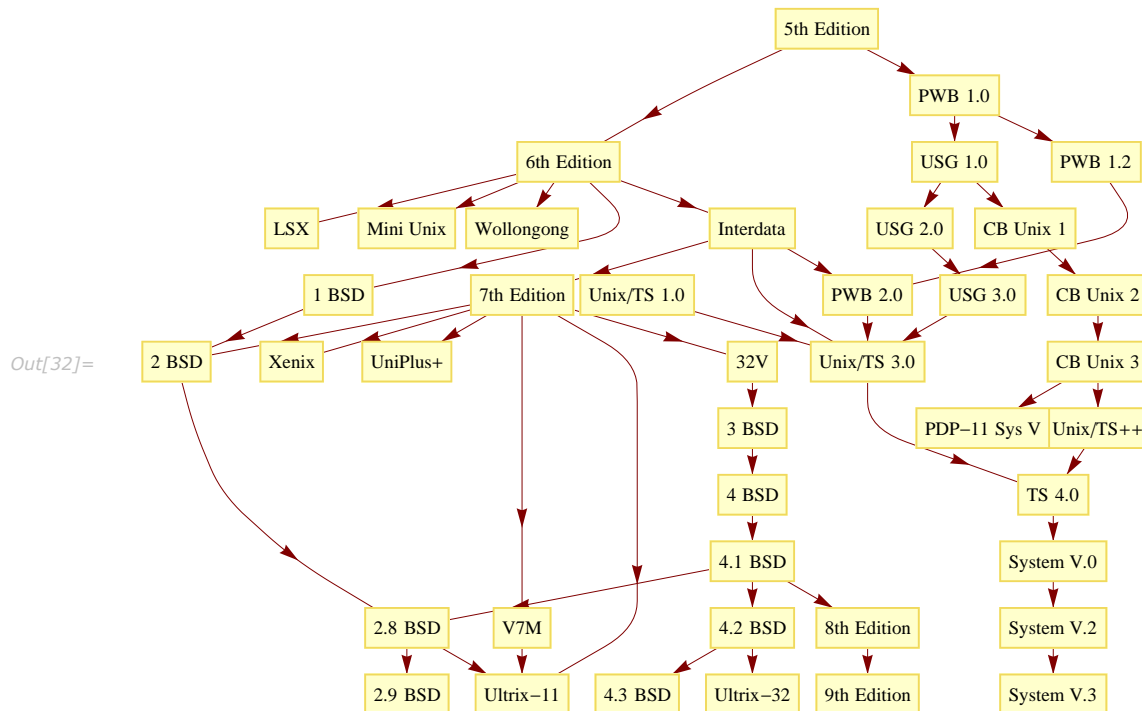


History of Unix

LayeredGraphPlot is suitable for visualizing historical events.

This shows a history of Unix.

```
In[32]:= LayeredGraphPlot[{"5th Edition" -> "6th Edition", "5th Edition" -> "PWB 1.0",
  "6th Edition" -> "1 BSD", "6th Edition" -> "Interdata",
  "6th Edition" -> "LSX", "6th Edition" -> "Mini Unix",
  "6th Edition" -> "Wollongong", "PWB 1.0" -> "PWB 1.2",
  "PWB 1.0" -> "USG 1.0", "1 BSD" -> "2 BSD", "Interdata" -> "PWB 2.0",
  "Interdata" -> "Unix/TS 3.0", "Interdata" -> "7th Edition",
  "PWB 1.2" -> "PWB 2.0", "USG 1.0" -> "USG 2.0", "USG 1.0" -> "CB Unix 1",
  "7th Edition" -> "2 BSD", "7th Edition" -> "32V", "7th Edition" -> "Xenix",
  "7th Edition" -> "Ultrix-11", "7th Edition" -> "UniPlus+",
  "7th Edition" -> "V7M", "PWB 2.0" -> "Unix/TS 3.0", "USG 2.0" -> "USG 3.0",
  "CB Unix 1" -> "CB Unix 2", "32V" -> "3 BSD",
  "Unix/TS 1.0" -> "Unix/TS 3.0", "USG 3.0" -> "Unix/TS 3.0",
  "CB Unix 2" -> "CB Unix 3", "3 BSD" -> "4 BSD", "V7M" -> "Ultrix-11",
  "Unix/TS 3.0" -> "TS 4.0", "CB Unix 3" -> "Unix/TS++",
  "CB Unix 3" -> "PDP-11 Sys V", "4 BSD" -> "4.1 BSD",
  "Unix/TS++" -> "TS 4.0", "4.1 BSD" -> "8th Edition", "4.1 BSD" -> "4.2 BSD",
  "4.1 BSD" -> "2.8 BSD", "2 BSD" -> "2.8 BSD", "TS 4.0" -> "System V.0",
  "4.2 BSD" -> "4.3 BSD", "4.2 BSD" -> "Ultrix-32", "2.8 BSD" -> "2.9 BSD",
  "2.8 BSD" -> "Ultrix-11", "System V.0" -> "System V.2",
  "8th Edition" -> "9th Edition", "System V.2" -> "System V.3"},
  VertexLabeling -> True, AspectRatio -> 0.7, PlotRangePadding -> Automatic]
```



Tree Drawing

`TreePlot` lays out the vertices of a graph in a tree of successive layers, or a collection of trees. If the graph g is not a tree, `TreePlot` lays out its vertices on the basis of a spanning tree of each component of the graph.

<code>TreePlot [{ { $v_{i1} \rightarrow v_{j1}$, $v_{i2} \rightarrow v_{j2}$, ... } }</code>	generate a tree plot of the graph in which vertex v_{ik} is connected to vertex v_{jk}
<code>TreePlot [{ { $v_{i1} \rightarrow v_{j1}$, lbl_1 } , ... } }</code>	associate labels lbl_k with edges in the graph
<code>TreePlot [g, pos]</code>	place roots of trees in the plot at position pos
<code>TreePlot [g, pos, v_k]</code>	use vertex v_k as the root node in the tree plot
<code>TreePlot [m]</code>	generate a layered plot of the graph represented by the adjacency matrix m

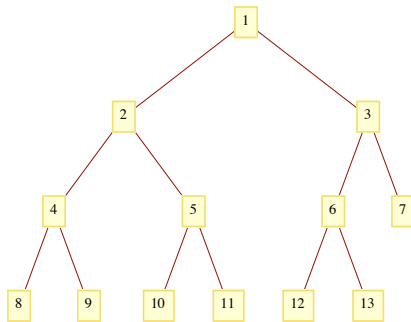
Tree drawing.

A simple graph and its tree plot.

```
In[1]:= g = {1 → 2, 2 → 4, 3 → 6, 4 → 8, 5 → 10, 6 → 12, 1 → 3, 2 → 5, 3 → 7, 4 → 9, 5 → 11, 6 → 13};
```

```
In[2]:= TreePlot[g, VertexLabeling -> True]
```

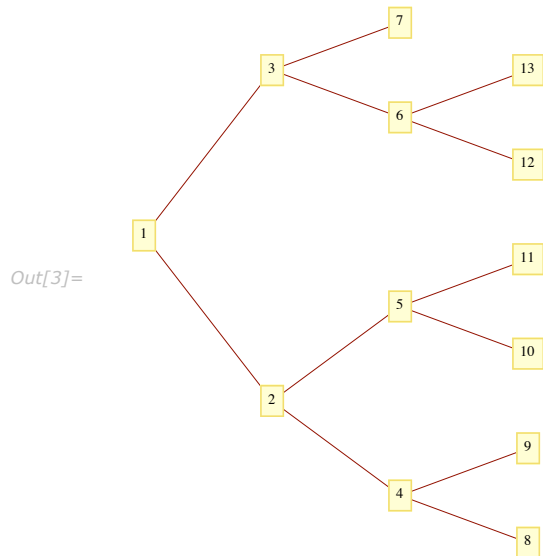
Out[2]=



By default, `TreePlot` places each tree root at the top. `TreePlot [g , pos]` places the roots at position pos . Possible positions are: Top, Bottom, Left, Right, and Center.

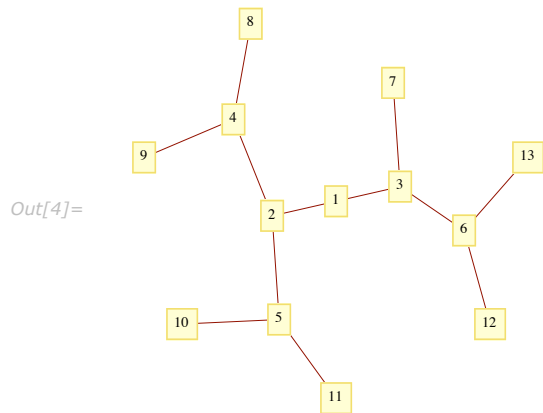
This plots the tree by placing the root left.

```
In[3]:= TreePlot[g, Left, VertexLabeling -> True]
```



This places the root at the center.

```
In[4]:= TreePlot[g, Center, VertexLabeling -> True]
```



Options for TreePlot

In addition to options for `Graphics`, the following options are accepted for `LayeredGraphPlot`.

<i>option name</i>	<i>default value</i>	
<code>DataRange</code>	<code>Automatic</code>	the range of vertex coordinates to generate
<code>DirectedEdges</code>	<code>True</code>	whether to show edges as directed arrows
<code>EdgeLabeling</code>	<code>True</code>	whether to include labels given for edges
<code>EdgeRenderingFunction</code>	<code>Automatic</code>	function to give explicit graphics for edges
<code>LayerSizeFunction</code>	<code>1&</code>	the height to allow for each layer
<code>MultiedgeStyle</code>	<code>Automatic</code>	how to draw multiple edges between vertices
<code>PackingMethod</code>	<code>Automatic</code>	method to use for packing components
<code>PlotRangePadding</code>	<code>Automatic</code>	how much padding to put around the plot
<code>PlotStyle</code>	<code>Automatic</code>	style in which objects are drawn
<code>SelfLoopStyle</code>	<code>Automatic</code>	how to draw edges linking a vertex to itself
<code>VertexCoordinateRules</code>	<code>Automatic</code>	rules for explicit vertex coordinates
<code>VertexLabeling</code>	<code>Automatic</code>	whether to show vertex names as labels
<code>VertexRenderingFunction</code>	<code>Automatic</code>	function to give explicit graphics for vertices

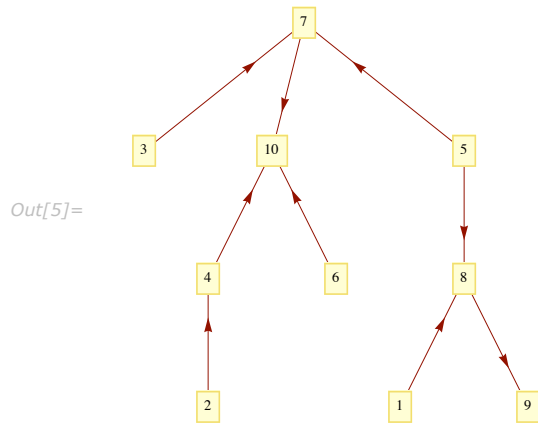
Options for `TreePlot`.

DirectedEdges

The option `DirectedEdges` specifies whether to draw edges as directed arrows. Possible values for this option are `True` or `False`. The default value for this option is `False`.

This shows a graph with edges represented by arrows instead of lines.

```
In[5]:= TreePlot[{1 → 8, 2 → 4, 3 → 7, 4 → 10, 5 → 7, 5 → 8, 6 → 10, 7 → 10, 8 → 9},
  DirectedEdges → True, VertexLabeling → True]
```

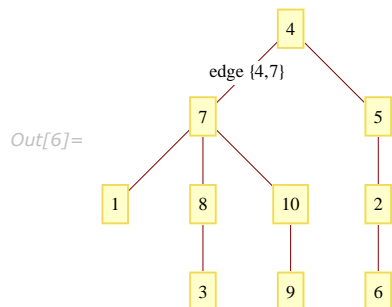


EdgeLabeling

The option `EdgeLabeling` specifies whether and how to display labels given for the edges. Possible values for this option are `True`, `False`, or `Automatic`. The default value for this option is `True`, which displays the supplied edge labels on the graph. With `EdgeLabeling -> Automatic`, the labels are shown as tooltips.

This displays the specified edge label.

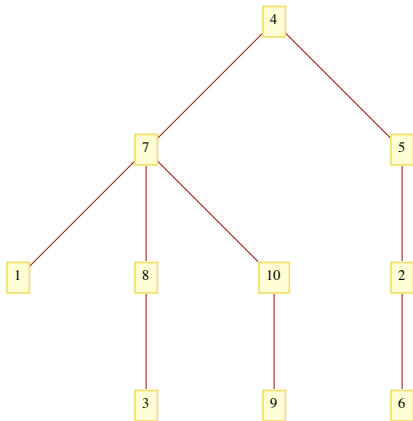
```
In[6]:= TreePlot[{1 → 7, 2 → 5, 2 → 6, 3 → 8, 4 → 5,
  {4 → 7, "edge {4,7}"}, 7 → 8, 7 → 10, 9 → 10}, VertexLabeling → True]
```



This displays the edge label as a tooltip. Place the cursor over the edge between vertices 4 and 7 to see the tooltip.

```
In[7]:= TreePlot[{1 → 7, 2 → 5, 2 → 6, 3 → 8, 4 → 5, {4 → 7, "edge {4,7}"}, 7 → 8,
  7 → 10, 9 → 10}, EdgeLabeling → Automatic, VertexLabeling → True]
```

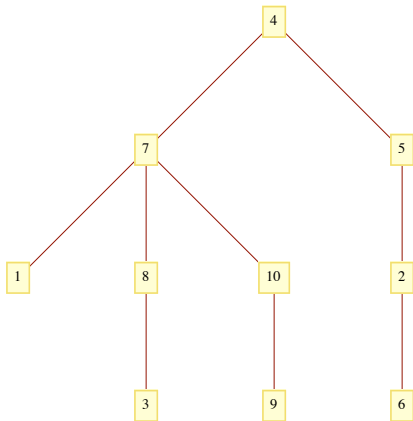
Out[7]=



This displays the labels as tooltips. Place the cursor over the edge between vertices 4 and 7 to see the tooltip.

```
In[8]:= TreePlot[{1 → 7, 2 → 5, 2 → 6, 3 → 8, 4 → 5,
  Tooltip[4 → 7, "edge {4,7}"], 7 → 8, 7 → 10, 9 → 10}, VertexLabeling → True]
```

Out[8]=



EdgeRenderingFunction

The option `EdgeRenderingFunction` specifies graphical representation of the graph edges. Possible values for this option are `Automatic`, `None`, or a function that gives a proper combination of graphics primitives and directives. With the default setting of `Automatic`, a dark red line is drawn for each edge. With `EdgeRenderingFunction -> None`, edges are not drawn.

This draws vertices only.

```
In[9]:= TreePlot[Table[1, {10}], {10}, EdgeRenderingFunction -> None, VertexLabeling -> True]
```

1

Out[9]=

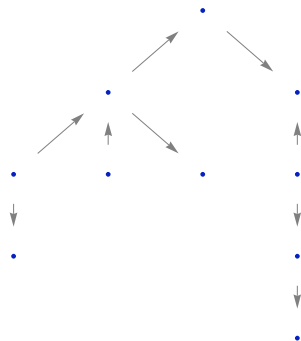
2 3 4 5 6 7 8 9 10

With `EdgeRenderingFunction -> g`, each edge is rendered with the graphics primitives and directives given by the function `g` that can take three or more arguments, in the form `g[{ r_i, \dots, r_j }, { v_i, v_j }, lbl_{ij}, \dots]`, where r_i, r_j are the coordinates of the beginning and ending points of the edge, v_i, v_j are the beginning and ending vertices, and lbl_{ij} is any label specified for the edge or `None`. Explicit settings for `EdgeRenderingFunction -> g` override settings for `EdgeLabeling` and `DirectedEdges`.

This plots edges as gray arrows with ends set back from vertices by a distance 0.3 (in the graph's coordinate system).

```
In[10]:= TreePlot[{1 -> 4, 1 -> 5, 2 -> 4, 3 -> 6, 3 -> 9, 4 -> 8, 4 -> 10, 6 -> 7, 8 -> 9},
EdgeRenderingFunction -> ({GrayLevel[0.5], Arrow[#1, 0.3]} &)]
```

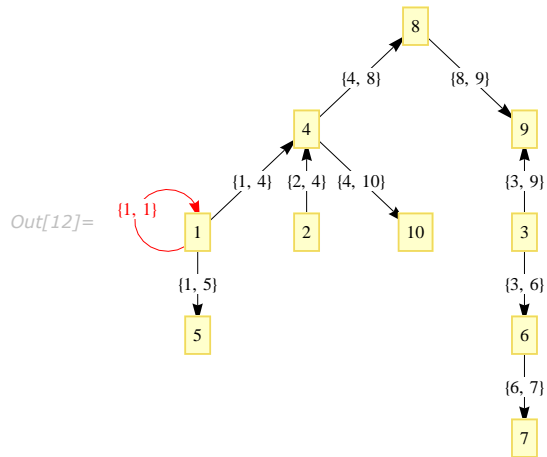
Out[10]=



This displays edges and self-loops with black and red arrows, respectively. The function `LineScaledCoordinate` from the Graph Utilities Package adds text at 50% along arrows.

```
In[11]:= << GraphUtilities`
```

```
In[12]:= TreePlot[{1 → 4, 1 → 1, 1 → 5, 2 → 4, 3 → 6, 3 → 9, 4 → 8, 4 → 10, 6 → 7, 8 → 9},
  EdgeRenderingFunction →
  ({If[First[#2] === Last[#2], Red, Black], Arrow[#1, .1], Text[#2,
    LineScaledCoordinate[#1, .5], Background → White]} &), VertexLabeling → True]
```



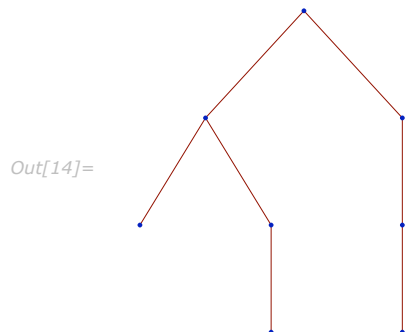
LayerSizeFunction

The `LayerSizeFunction` option specifies the relative height to allow for each layer. By default the height is 1. Possible values include a function that gives real machine numbers.

This defines and plots a tree.

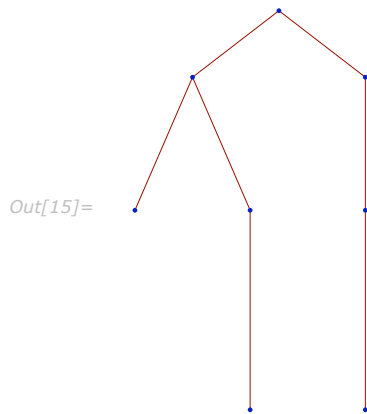
```
In[13]:= g = {1 → 4, 2 → 6, 2 → 7, 2 → 8, 3 → 8, 4 → 5, 5 → 6};
```

```
In[14]:= TreePlot[g]
```



This plots the same tree, with the first layer a relative height of 1, the second 2, and the third 3.

```
In[15]:= TreePlot[g, LayerSizeFunction -> (# &)]
```

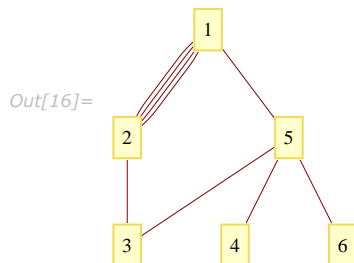


MultiedgeStyle

The option `MultiedgeStyle` specifies whether to draw multiple edges between two vertices. Possible values for `MultiedgeStyle` are `Automatic` (the default), `True`, `False`, or a positive real number. With the default setting `MultiedgeStyle -> Automatic`, multiple edges are shown for a graph specified by a list of rules, but not shown if specified by an adjacency matrix. With `MultiedgeStyle -> δ` , the multiedges are spread out to a scaled distance of δ .

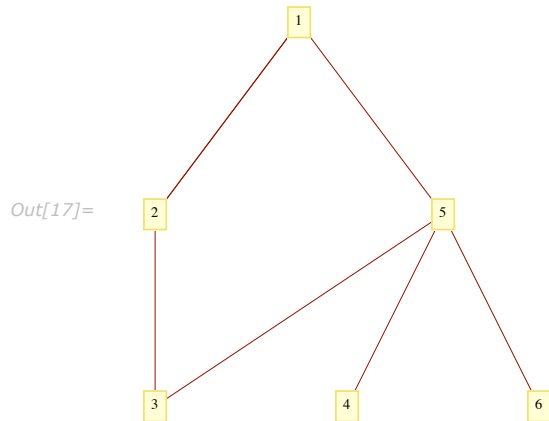
By default, multiple edges are shown if a graph is given as a list of rules.

```
In[16]:= TreePlot[{1 -> 2, 2 -> 1, 1 -> 2, 1 -> 2, 2 -> 3, 3 -> 5, 4 -> 5, 5 -> 6, 5 -> 1},
  VertexLabeling -> True]
```



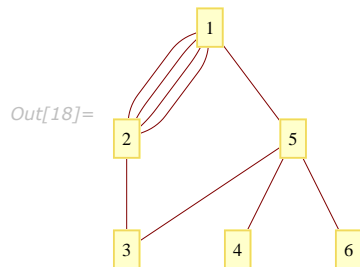
But multiple edges are not shown for graphs specified by an adjacency matrix.

```
In[17]:= TreePlot[ $\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ , VertexLabeling -> True]
```



This spreads multiple edges by the specified amount.

```
In[18]:= TreePlot[{1 -> 2, 2 -> 1, 1 -> 2, 1 -> 2, 2 -> 3, 3 -> 5, 4 -> 5, 5 -> 6, 5 -> 1},
MultiedgeStyle -> 0.25, VertexLabeling -> True]
```

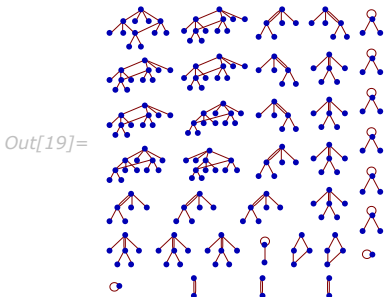


PackingMethod

The option `PackingMethod` specifies the method used for packing disconnected components. Possible values for the option are `Automatic` (the default), `"ClosestPacking"`, `"ClosestPackingCenter"`, `"Layered"`, `"LayeredLeft"`, `"LayeredTop"`, and `"NestedGrid"`. With `PackingMethod -> "ClosestPacking"`, components are packed as close together as possible using a polyomino method [6], starting from the top left. With `PackingMethod -> "ClosestPackingCenter"`, components are packed starting from the center. With `PackingMethod -> "Layered"`, components are packed in layers starting from the top left. With `PackingMethod -> "LayeredLeft"` or `PackingMethod -> "LayeredTop"`, components are packed in layers starting from the top or left respectively. With `PackingMethod -> "NestedGrid"`, components are arranged in a nested grid. The typical effective default setting is `PackingMethod -> "Layered"`, and the packing starts with components of the largest bounding box area.

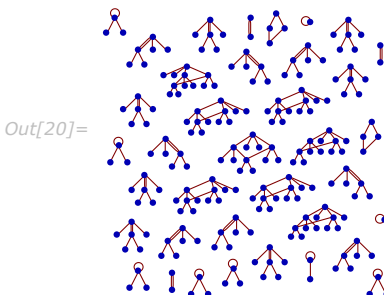
This shows the packing of disconnected components by the default method.

```
In[19]:= TreePlot[Table[i -> Mod[i^3, 221], {i, 0, 221}]]
```



This shows the packing of disconnected components using the `"ClosestPackingCenter"` method.

```
In[20]:= TreePlot[Table[i -> Mod[i^3, 221], {i, 0, 221}],  
PackingMethod -> "ClosestPackingCenter"]
```



PlotRangePadding

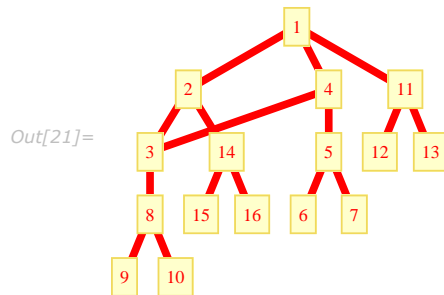
PlotRangePadding is a common option for graphics functions inherited by TreePlot.

PlotStyle

PlotStyle is a common option for graphics functions inherited by TreePlot. The option PlotStyle specifies the style in which objects are drawn.

Draw edges with thicker lines, and draw both edges and vertex labels in red.

```
In[21]:= TreePlot[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 6 → 5, 7 → 5, 5 → 4, 9 → 8,
  10 → 8, 8 → 3, 12 → 11, 13 → 11, 11 → 1, 15 → 14, 16 → 14, 14 → 2},
  VertexLabeling → True, PlotStyle → {Red, Thickness[0.02]}]
```

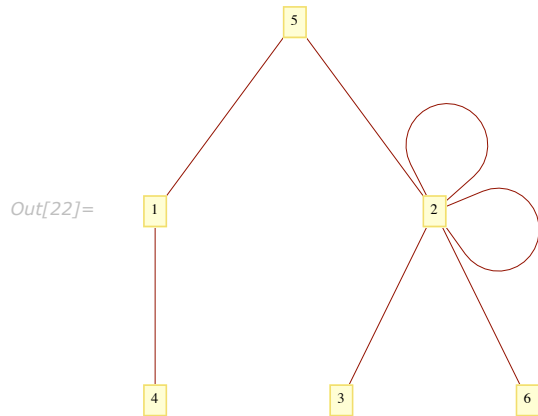


SelfLoopStyle

The option SelfLoopStyle specifies whether and how to draw loops for vertices that are linked to themselves. Possible values for the option are Automatic (the default), True, False, or a positive real number. With SelfLoopStyle -> Automatic, self-loops are shown if the graph is specified by a list of rules, but not if it is specified by an adjacency matrix. With SelfLoopStyle -> δ , the self-loops are drawn with a diameter of δ (relative to the average edge length).

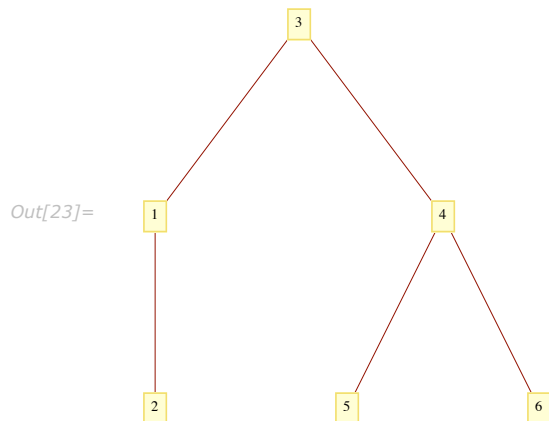
By default, self-loops are displayed for a graph specified by a list of rules.

```
In[22]:= TreePlot[{1 → 4, 1 → 5, 2 → 3, 2 → 5, 2 → 6, 2 → 2, 2 → 2}, VertexLabeling → True]
```



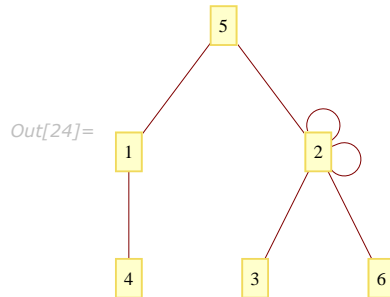
Self-loops are not shown if the graph is specified by an adjacency matrix.

```
In[23]:= TreePlot[ $\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ , VertexLabeling → True]
```



This shows self-loops whose diameters equal 0.3 times the average length of the edges.

```
In[24]:= TreePlot[{1 → 4, 1 → 5, 2 → 3, 2 → 5, 2 → 6, 2 → 2, 2 → 2},
  VertexLabeling → True, SelfLoopStyle → 0.3]
```

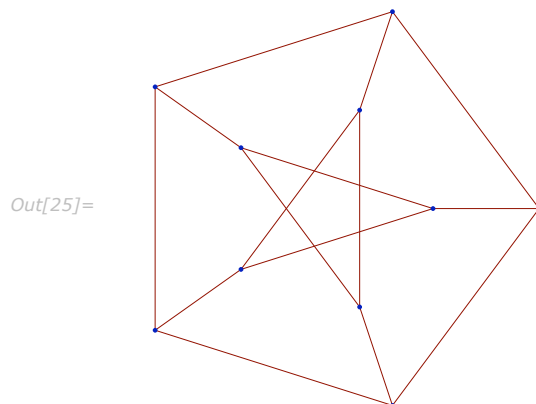


VertexCoordinateRules

The option `VertexCoordinateRules` specifies the coordinates of the vertices. Possible values are `None` or a list of coordinates. Coordinates specified by a list of rules are not supported by `TreePlot` currently.

This draws the Petersen graph using known coordinates.

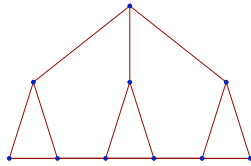
```
In[25]:= TreePlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7,
  7 → 8, 8 → 9, 9 → 10, 6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10},
  VertexCoordinateRules → {{0.30901699437494745`, 0.9510565162951535`},
  {-0.8090169943749476`, -0.587785252292473`}, {0.30901699437494723`,
  -0.9510565162951536`}, {-0.8090169943749473`, 0.5877852522924732`},
  {1., 0}, {0.6180339887498949`, 1.902113032590307`}, {-1.6180339887498947`,
  1.1755705045849465`}, {-1.6180339887498951`, -1.175570504584946`},
  {0.6180339887498945`, -1.9021130325903073`}, {2., 0}}]
```



This draws with the default method.

```
In[26]:= TreePlot[{1 → 3, 1 → 4, 2 → 4, 2 → 5, 3 → 5, 6 → 7,
  7 → 8, 8 → 9, 9 → 10, 6 → 10, 1 → 6, 2 → 7, 3 → 8, 4 → 9, 5 → 10}]
```

Out[26]=



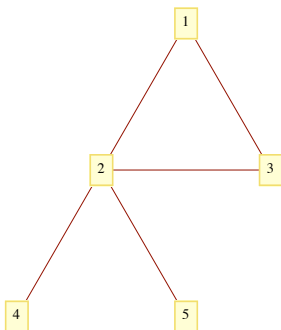
VertexLabeling

The option `VertexLabeling` specifies whether to show vertex names as labels. Possible values for this option are `True`, `False`, `Automatic` (the default) and `Tooltip`. `VertexLabeling -> True` shows the labels. For graphs specified by an adjacency matrix, vertex labels are taken to be successive integers $1, 2, \dots, n$, where n is the size of the matrix. For graphs specified by a list of rules, labels are the expressions used in the rules. `VertexLabeling -> False` displays each vertex as a point. `VertexLabeling -> Tooltip` displays each vertex as a point, but gives its name in a tooltip. `VertexLabeling -> Automatic` displays each vertex as a point, giving its name in a tooltip if the number of vertices is not too large. You can also use `Tooltip[v_k , v_{lbl}]` anywhere in the list of rules to specify an alternative tooltip for a vertex v_k .

This draws the graph with labels given as indices of the adjacency matrix.

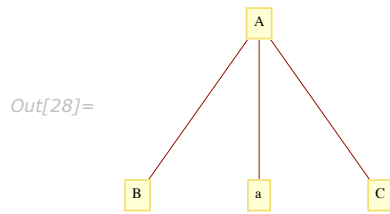
```
In[27]:= TreePlot[ $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$ , VertexLabeling -> True]
```

Out[27]=



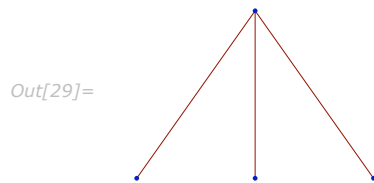
This uses the labels specified in the list of rules.

```
In[28]:= TreePlot[{"A" → "B", "A" → "a", "C" → "A"}, VertexLabeling → True]
```



This plots vertices as points, and displays vertex names in tooltips. Place the cursor above the vertices to see the labels.

```
In[29]:= TreePlot[{"A" → "B", "A" → "a", "C" → "A"}, VertexLabeling → Tooltip]
```



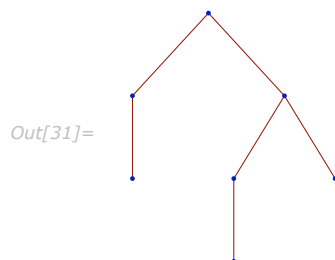
VertexRenderingFunction

The option `VertexRenderingFunction` specifies graphical representation of the graph edges. Possible values for this option are `Automatic`, `None`, or a function that gives a proper combination of graphics primitives and directives. With the default setting of `Automatic`, vertices are displayed as points, with their names given in tooltips.

By default, vertices are displayed as points and, for small graphs, labeled in tooltips. Point the cursor at a vertex to see the tooltip.

```
In[30]:= g = {1 → 3, 1 → 4, 2 → 3, 2 → 5, 2 → 6, 5 → 7};
```

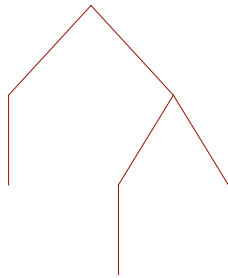
```
In[31]:= TreePlot[g]
```



This draws the same graph, but without the vertices.

```
In[32]:= TreePlot[g, VertexRenderingFunction -> None]
```

Out[32]=

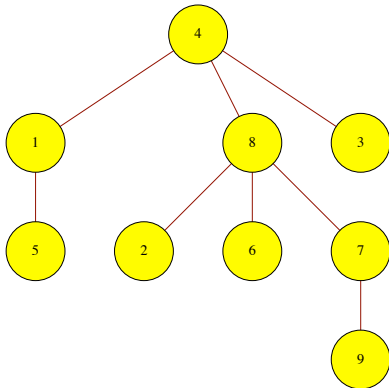


With `VertexRenderingFunction -> g`, each vertex is rendered with the graphics primitives given by $g[r_i, v_i, \dots]$, where r_i is the coordinate of the vertex and v_i is the label of the vertex. Explicit settings for `VertexRenderingFunction -> g` override settings for `VertexLabeling`.

This shows vertices as yellow disks.

```
In[33]:= TreePlot[{1 -> 4, 1 -> 5, 2 -> 8, 3 -> 4, 4 -> 8, 6 -> 8, 7 -> 8, 7 -> 9},  
VertexRenderingFunction ->  
({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]
```

Out[33]=



Example Gallery

k-ary tree

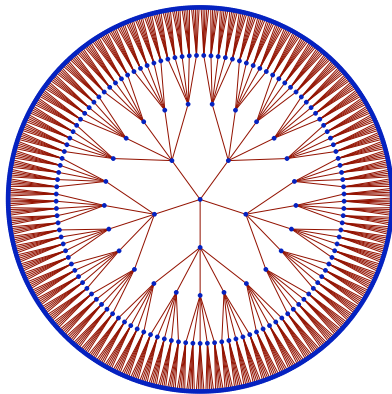
This defines a *k*-ary tree.

```
In[34]:= KaryTree[level_, k_ : 2] := Flatten[Table[Table[i → k * i + j, {j, -(k - 2), 1, 1}],
{ i, (k^level - 1) / (k - 1) }]] /; (level ≥ 1 && k > 1);
```

This plots a 4-ary tree of 4 levels.

```
In[35]:= TreePlot[KaryTree[4, 5], Center]
```

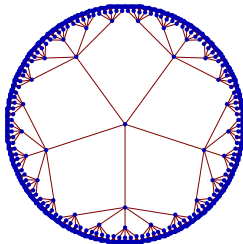
Out[35]=



This plots the same graph, but with the height of each level *i* proportional to $1/i^2$.

```
In[36]:= TreePlot[KaryTree[4, 5], Center, LayerSizeFunction → (1 / #^2 &)]
```

Out[36]=



This sets the height of each level i proportional to $(-0.5)^i$.

```
In[37]:= TreePlot[KaryTree[4, 5], Center, LayerSizeFunction → ((-0.5) ^ # &)]
```

Out[37]=

