DiscreteMath‘Combinatorica‘

`DiscreteMath‘Combinatorica‘` extends *Mathematica* by over 450 functions in combinatorics and graph theory. It includes functions for constructing graphs and other combinatorial objects, computing invariants of these objects, and finally displaying them. This documentation covers only a subset of these functions. The best guide to this package is the book *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, by Steven Skiena and Sriram Pemmaraju, published by Cambridge University Press, 2003. The new *Combinatorica* is a substantial rewrite of the original 1990 version. It is now much faster than before, and provides improved graphics and significant additional functionality.

We encourage you to visit our website, www.combinatorica.com, where you will find the latest release of the package, an editor for *Combinatorica* graphs, and additional files of interest.

This loads the package.

```
In[1]:= <<DiscreteMath‘Combinatorica‘
```

Permutations and Combinations

Permutations and subsets are the most basic combinatorial objects. `DiscreteMath‘Combinatorica‘` provides functions for constructing objects both randomly and deterministically, to rank and unrank them, and to compute invariants on them. Here we provide examples of some of these functions in action.

These permutations are generated in minimum change order, where successive permutations differ by exactly one transposition. The built-in generator `Permutations` constructs permutations in lexicographic order.

```
In[2]:= MinimumChangePermutations[{a,b,c}]

Out[2]= {{a, b, c}, {b, a, c}, {c, a, b}, {a, c, b}, {b, c, a}, {c, b, a}}
```

The ranking function illustrates that the built-in function `Permutations` uses lexicographic sequencing.

```
In[3]:= Map[RankPermutation, Permutations[{1,2,3,4}]]

Out[3]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23}
```

With $(3!) = 6$ distinct permutations of three elements, within 20 random permutations we are likely to see all of them. Observe that it is unlikely for the first six permutations to all be distinct.

```
In[4]:= Table[RandomPermutation[3], {20}]

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

A fixed point of a permutation p is an element in the same position in p as in the inverse of p. Thus, the only fixed point in this permutation is 7.

```
In[5]:= InversePermutation[{4,8,5,2,1,3,7,6}]

Out[5]= {5, 4, 6, 1, 3, 8, 7, 2}
```

The identity permutation consists of n singleton cycles or fixed points.

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

Out[6]= {{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}}
```

The classic problem in Polya theory is counting how many different ways necklaces can be made out of k beads, when there are m different types or colors of beads to choose from. When two necklaces are considered the same if they can be obtained only by shifting the beads (as opposed to turning the necklace over), the symmetries are defined by k permutations, each of which is a cyclic shift of the identity permutation. When a variable is specified for the number of colors, a polynomial results.

```
In[7]:= NecklacePolynomial[8,m,Cyclic]

Out[7]= m/2 + m^2/4 + m^4/8 + m^8/8
```

The number of inversions in a permutation is equal to that of its inverse.

```
In[8]:= (p=RandomPermutation[50]; {Inversions[p], Inversions[InversePermutation[p]]})

Out[8]= {642, 642}
```

Generating subsets incrementally is efficient when the goal is to find the first subset with a given property, since every subset need not be constructed.

```
In[9]:= Table[UnrankSubset[n,{a,b,c,d}], {n,0,15}]

Out[9]= {{}, {d}, {c, d}, {c}, {b, c}, {b, c, d}, {b, d}, {b}, {a, b},
         {a, b, d}, {a, b, c, d}, {a, b, c}, {a, c}, {a, c, d}, {a, d}, {a}}
```

In a Gray code, each subset differs in exactly one element from its neighbors. Observe that the last eight subsets all contain 1, while none of the first eight do.

```
In[10]:= GrayCodeSubsets[{1,2,3,4}]

Out[10]= {{}, {4}, {3, 4}, {3}, {2, 3}, {2, 3, 4}, {2, 4}, {2}, {1, 2},
          {1, 2, 4}, {1, 2, 3, 4}, {1, 2, 3}, {1, 3}, {1, 3, 4}, {1, 4}, {1}}
```

A k-subset is a subset with exactly k elements in it. Since the lead element is placed in first, the k-subsets are given in lexicographic order.

```
In[11]:= KSubsets[{1,2,3,4,5},3]

Out[11]= {{1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 4},
          {1, 3, 5}, {1, 4, 5}, {2, 3, 4}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}}
```

BinarySearch	DerangementQ
Derangements	DistinctPermutations
EncroachingListSet	FromCycles
FromInversionVector	HeapSort
Heapify	HideCycles
IdentityPermutation	Index
InversePermutation	InversionPoset
Inversions	InvolutionQ
Involutions	Josephus
LexicographicPermutations	LongestIncreasingSubsequence
MinimumChangePermutations	NextPermutation
PermutationQ	PermutationType
PermutationWithCycle	Permute
RandomHeap	RandomPermutation
RankPermutation	RevealCycles
Runs	SelectionSort
SignaturePermutation	ToCycles
ToInversionVector	UnrankPermutation

`Combinatorica` functions for permutations.

BinarySubsets	DeBruijnSequence
GrayCodeKSubsets	GrayCodeSubsets
GrayGraph	KSubsets
LexicographicSubsets	NextBinarySubset
NextGrayCodeSubset	NextKSubset
NextLexicographicSubset	NextSubset
NthSubset	RandomKSubset
RandomSubset	RankBinarySubset
RankGrayCodeSubset	RankKSubset
RankSubset	Strings
Subsets	UnrankBinarySubset
UnrankGrayCodeSubset	UnrankKSubset
UnrankSubset	

`Combinatorica` functions for subsets.

AlternatingGroup	AlternatingGroupIndex
CycleIndex	CycleStructure
Cycles	Cyclic
CyclicGroup	CyclicGroupIndex
Dihedral	DihedralGroup
DihedralGroupIndex	EquivalenceClasses
KSubsetGroup	KSubsetGroupIndex
ListNecklaces	MultiplicationTable
NecklacePolynomial	OrbitInventory
OrbitRepresentatives	Orbits
Ordered	PairGroup
PairGroupIndex	PermutationGroupQ
SamenessRelation	SymmetricGroup
SymmetricGroupIndex	

`Combinatorica` functions for group theory.

Partitions, Compositions, and Young Tableaux

A partition of a positive integer n is a set of k strictly positive integers whose sum is n. A composition of n is a particular arrangement of nonnegative integers whose sum is n. A set partition of n elements is a grouping of all the elements into nonempty, nonintersecting subsets. A Young tableau is a structure of integers 1, …, n where the number of elements in each row is defined by an integer partition of n. Further, the elements of each row and column are in increasing order, and the rows are left justified. These four related combinatorial objects have a host of interesting applications and properties.

Here are the eleven partitions of 6. Observe that they are given in reverse lexicographic order.

```
In[12]:= Partitions[6]

Out[12]= {{6}, {5, 1}, {4, 2}, {4, 1, 1}, {3, 3}, {3, 2, 1},
          {3, 1, 1, 1}, {2, 2, 2}, {2, 2, 1, 1}, {2, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1}}
```

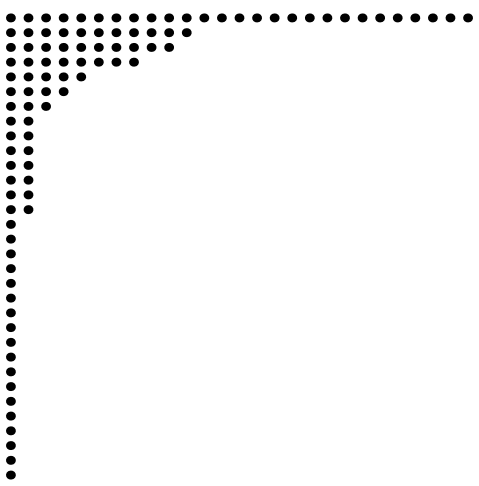
Although the number of partitions grows exponentially, it does so more slowly than permutations or subsets, so complete tables can be generated for larger values of n.

```
In[13]:= Length[Partitions[20]]

Out[13]= 627
```

Ferrers diagrams represent partitions as patterns of dots. They provide a useful tool for visualizing partitions, because moving the dots around provides a mechanism for proving bijections between classes of partitions. Here we construct a random partition of 100.

```
In[14]:= FerrersDiagram[RandomPartition[100]]
```



```
Out[14]= -Graphics-
```

Here every composition of 5 into 3 parts is generated exactly once.

```
In[15]:= Compositions[5,3]

Out[15]= {{0, 0, 5}, {0, 1, 4}, {0, 2, 3}, {0, 3, 2}, {0, 4, 1}, {0, 5, 0}, {1, 0, 4},
          {1, 1, 3}, {1, 2, 2}, {1, 3, 1}, {1, 4, 0}, {2, 0, 3}, {2, 1, 2}, {2, 2, 1},
          {2, 3, 0}, {3, 0, 2}, {3, 1, 1}, {3, 2, 0}, {4, 0, 1}, {4, 1, 0}, {5, 0, 0}}
```

Set partitions are different than integer partitions, representing the ways we can partition distinct elements into subsets. They are useful for representing colorings and clusterings.

```
In[16]:= SetPartitions[3]

Out[16]= {{{1, 2, 3}}, {{1}, {2, 3}}, {{1, 2}, {3}}, {{1, 3}, {2}}, {{1}, {2}, {3}}}
```

The list of tableaux of shape $\{2, 2, 1\}$ illustrates the amount of freedom available to tableaux structures. The smallest element is always in the upper left-hand corner, but the largest element is free to be the rightmost position of the last row defined by the *distinct* parts of the partition.

```
In[17]:= Tableaux[{2,2,1}]

Out[17]= {{{1, 4}, {2, 5}, {3}}, {{1, 3}, {2, 5}, {4}},
           {{1, 2}, {3, 5}, {4}}, {{1, 3}, {2, 4}, {5}}, {{1, 2}, {3, 4}, {5}}}
```

By iterating through the different integer partitions as shapes, all tableaux of a particular size can be constructed.

```
In[18]:= Tableaux[3]

Out[18]= {{{1, 2, 3}}, {{1, 3}, {2}}, {{1, 2}, {3}}, {{1}, {2}, {3}}}
```

The hook length formula can be used to count the number of tableaux for any shape. Using the hook length formula over all partitions of n computes the number of tableaux on n elements.

```
In[19]:= NumberOfTableaux[10]

Out[19]= 9496
```

Each of the 117,123,756,750 tableaux of this shape will be selected with equal likelihood.

```
In[20]:= TableForm[ RandomTableau[{6,5,5,4,3,2}] ]

Out[20]//TableForm=
1    2    3    4    16   17
5    6    12   19   21
7    8    13   20   24
9    10   18   22
11   14   23
15   25
```

A pigeonhole result states that any sequence of $n^2 + 1$ distinct integers must contain either an increasing or a decreasing scattered subsequence of length $n + 1$.

```
In[21]:= LongestIncreasingSubsequence[
           RandomPermutation[50] ]

Out[21]= {1, 6, 9, 11, 17, 19, 21, 25, 27, 33, 34, 35}
```

Compositions	DominatingIntegerPartitionQ
DominationLattice	DurfeeSquare
FerrersDiagram	NextComposition
NextPartition	PartitionQ
Partitions	RandomComposition
RandomPartition	TransposePartition

`Combinatorica` functions for integer partitions.

CoarserSetPartitionQ	FindSet
InitializeUnionFind	KSetPartitions
PartitionLattice	RGFQ
RGFToSetPartition	RGFs
RandomKSetPartition	RandomRGF
RandomSetPartition	RankKSetPartition
RankRGF	RankSetPartition
SetPartitionListViaRGF	SetPartitionQ
SetPartitionToRGF	SetPartitions
ToCanonicalSetPartition	UnionSet
UnrankKSetPartition	UnrankRGF
UnrankSetPartition	

`Combinatorica` functions for set partitions.

ConstructTableau	DeleteFromTableau
FirstLexicographicTableau	InsertIntoTableau
LastLexicographicTableau	NextTableau
PermutationToTableaux	RandomTableau
TableauClasses	TableauQ
Tableaux	TableauxToPermutation
TransposeTableau	

`Combinatorica` functions for Young tableaux.

Backtrack	BellB
Cofactor	Distribution
Element	Eulerian
NumberOf2Paths	NumberOfCompositions
NumberOfDerangements	NumberOfDirectedGraphs
NumberOfGraphs	NumberOfInvolutions
NumberOfKPaths	NumberOfNecklaces
NumberOfPartitions	NumberOfPermutationsByCycles
NumberOfPermutationsByInversions	NumberOfPermutationsByType
NumberOfSpanningTrees	NumberOfTableaux
StirlingFirst	StirlingSecond

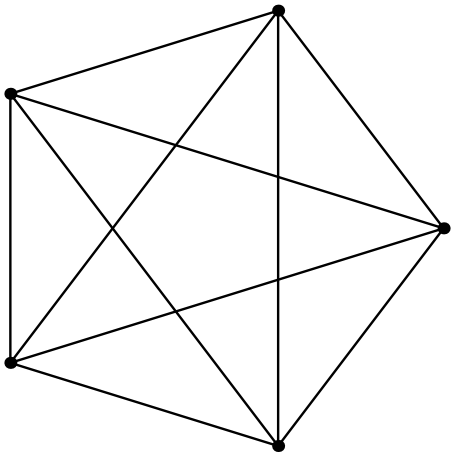
`Combinatorica` functions for counting.

Representing Graphs

We define a graph to be a set of vertices with a set of edges, where an edge is defined as a pair of vertices. The representation of graphs takes on different requirements depending upon whether the intended consumer is a person or a machine. Computers digest graphs best as data structures such as adjacency matrices or lists. People prefer a visualization of the structure as a collection of points connected by lines, which implies adding geometric information to the graph.

In the complete graph on five vertices, denoted K_5 , each vertex is adjacent to all other vertices. `CompleteGraph[n]` constructs the complete graph on n vertices.

```
In[22]:= ShowGraph[ CompleteGraph[5] ];
```



The internals of the graph representation are not shown to the user—only a notation with the number of edges and vertices, followed by whether the graph is directed or undirected.

```
In[23]:= CompleteGraph[5]

Out[23]= -Graph:<10, 5, Undirected>-
```

The adjacency matrix of K_5 shows that each vertex is adjacent to all other vertices. The main diagonal consists of zeros, since there are no self-loops in the complete graph, meaning edges from a vertex to itself.

```
In[24]:= TableForm[ToAdjacencyMatrix[CompleteGraph[5]]]

Out[24]//TableForm=
0  1  1  1  1
1  0  1  1  1
1  1  0  1  1
1  1  1  0  1
1  1  1  1  0
```

The standard embedding of K_5 consists of five vertices equally spaced on a circle.

```
In[25]:= Vertices[ CompleteGraph[5] ]

Out[25]= {{0.309017, 0.951057}, {-0.809017, 0.587785},
          {-0.809017, -0.587785}, {0.309017, -0.951057}, {1., 0}}
```

The number of vertices in a graph is termed the order of the graph.

```
In[26]:= V[ CompleteGraph[5] ]

Out[26]= 5
```

`M` returns the number of edges in a graph.

```
In[27]:= M[ CompleteGraph[5] ]

Out[27]= 10
```

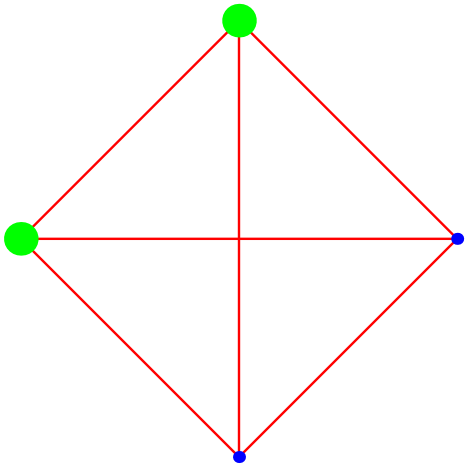
Edge/vertex colors/styles can be globally modified, giving complete flexibility to change the appearance of a graph.

```
In[28]:= g = SetGraphOptions[CompleteGraph[4], VertexColor -> Red,
              EdgeColor -> Blue]

Out[28]= -Graph:<6, 4, Undirected>-
```

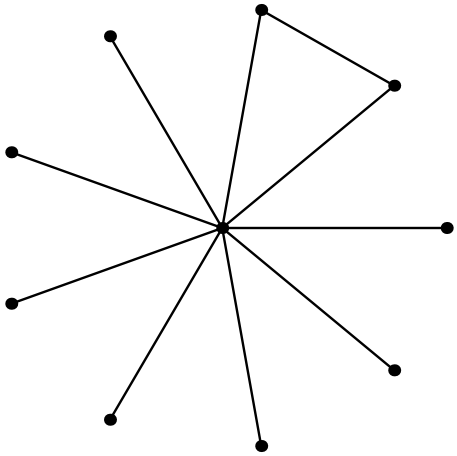
The colors, styles, labels, and weights of individual vertices and edges can also be changed individually, perhaps to highlight interesting features of the graph.

```
In[29]:= ShowGraph[ SetGraphOptions[ CompleteGraph[4],
                   {{1, 2, VertexColor -> Green, VertexStyle -> Disk[Large]},
                    {3, 4, VertexColor -> Blue}},
                   EdgeColor -> Red
            ] ];
```



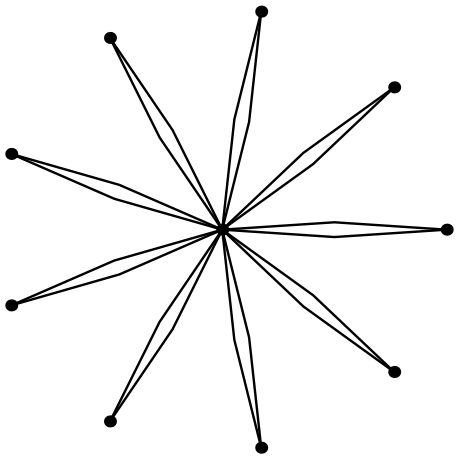
A star is a tree with one vertex of degree $n-1$. Adding any new edge to a star produces a cycle of length 3.

In[30]:= **ShowGraph[AddEdge[Star[10], {1,2}]];**



Graphs with multi-edges and self-loops are supported. Here there are two copies of each edge of a star.

In[31]:= **ShowGraph[GraphSum[Star[10], Star[10]]];**



The adjacency list representation of a graph consists of n lists, one list for each vertex v_i, $1 \le i \le n$, which records the vertices to which v_i is adjacent. Each vertex in the complete graph is adjacent to all other vertices.

In[32]:= **TableForm[ToAdjacencyLists[CompleteGraph[5]]]**

Out[32]//TableForm=

2	3	4	5
1	3	4	5
1	2	4	5
1	2	3	5
1	2	3	4

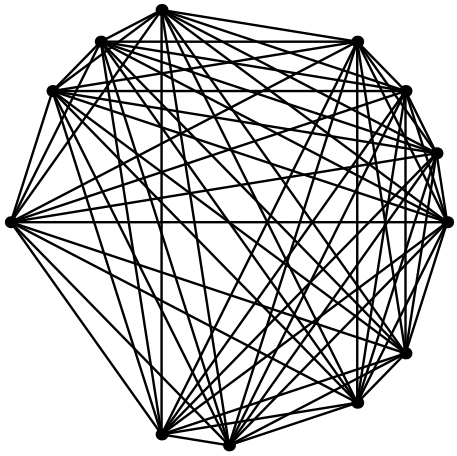
There are $n(n-1)$ ordered pairs of edges defined by a complete graph of order n.

```
In[33]:= ToOrderedPairs[ CompleteGraph[5] ]

Out[33]= {{2, 1}, {3, 1}, {4, 1}, {5, 1}, {3, 2}, {4, 2}, {5, 2}, {4, 3}, {5, 3}, {5, 4},
          {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}}
```

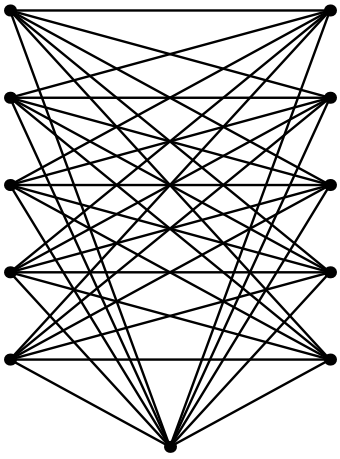
An induced subgraph of a graph G is a subset of the vertices of G together with any edges whose endpoints are both in this subset. An induced subgraph that is complete is called a clique. Any subset of the vertices in a complete graph defines a clique.

```
In[34]:= ShowGraph[ InduceSubgraph[CompleteGraph[20],
                   RandomSubset[Range[20]]] ];
```



The vertices of a bipartite graph have the property that they can be partitioned into two sets such that no edge connects two vertices of the same set. Contracting an edge in a bipartite graph can ruin its bipartiteness. Note the self-loop created by the contraction.

```
In[35]:= ShowGraph[ Contract[ CompleteGraph[6,6],{1,7} ] ];
```



A breadth-first search of a graph explores all the vertices adjacent to the current vertex before moving on. A breadth-first traversal of a simple cycle alternates sides as it wraps around the cycle.

```
In[36]:= BreadthFirstTraversal[Cycle[20],1]

Out[36]= {1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 19, 17, 15, 13, 11, 9, 7, 5, 3}
```

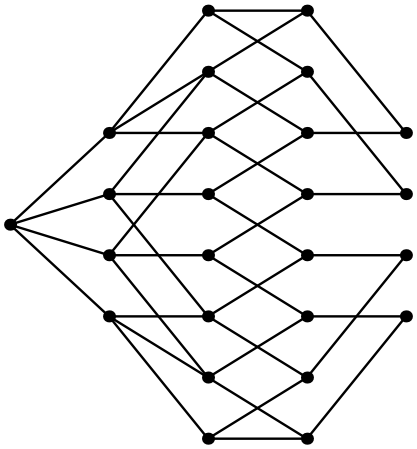
In a depth-first search, the children of the first son of a vertex are explored before visiting his brothers. The depth-first traversal differs from the breadth-first traversal above in that it proceeds directly around the cycle.

```
In[37]:= DepthFirstTraversal[Cycle[20], 1]

Out[37]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

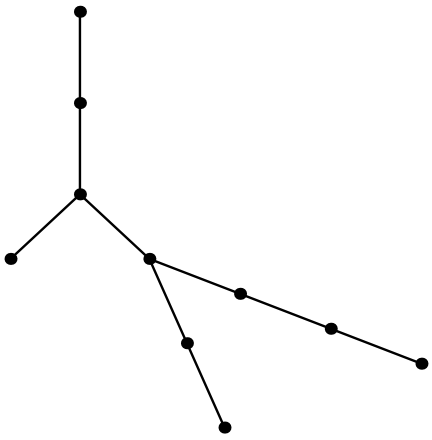
Different drawings or embeddings of a graph can reveal different aspects of its structure. The default embedding for a grid graph is a ranked embedding from all the vertices on one side. Ranking from the center vertex yields a different but interesting drawing.

```
In[38]:= ShowGraph[
           RankedEmbedding[GridGraph[5,5],{13}]];
```



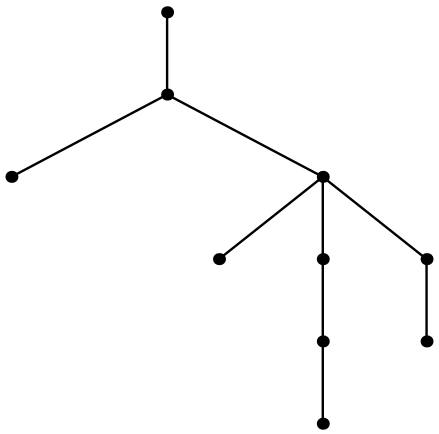
The radial embedding of a tree is guaranteed to be planar, but radial embeddings can be used with any graph. Here we see a radial embedding of a random labeled tree.

```
In[39]:= ShowGraph[ RandomTree[10] ];
```



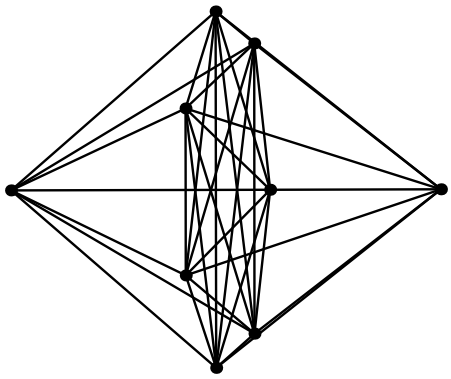
By arbitrarily selecting a root, any tree can be represented as a rooted tree.

In[40]:=
```
ShowGraph[ RootedEmbedding[RandomTree[10],1] ];
```



An interesting general heuristic for drawing graphs models the graph as a system of springs and lets Hooke's law space the vertices. Here it does a good job illustrating the `Join` operation, where each vertex of K_7 is connected to each of two disconnected vertices. In achieving the minimum energy configuration, these two vertices end up on different sides of K_7 .

In[41]:=
```
ShowGraph[
          SpringEmbedding[
          GraphJoin[EmptyGraph[2], CompleteGraph[7]]]];
```



AddEdge	AddEdges
AddVertex	AddVertices
ChangeEdges	ChangeVertices
Contract	DeleteCycle
DeleteEdge	DeleteEdges
DeleteVertex	DeleteVertices
InduceSubgraph	MakeDirected
MakeSimple	MakeUndirected
PermuteSubgraph	RemoveMultipleEdges
RemoveSelfLoops	ReverseEdges

`Combinatorica` functions for modifying graphs.

Edges	FromAdjacencyLists
FromAdjacencyMatrix	FromOrderedPairs
FromUnorderedPairs	IncidenceMatrix
ToAdjacencyLists	ToAdjacencyMatrix
ToOrderedPairs	ToUnorderedPairs

`Combinatorica` functions for graph format translation.

Algorithm	Box
Brelaz	Center
Circle	Directed
Disk	EdgeColor
EdgeDirection	EdgeLabel
EdgeLabelColor	EdgeLabelPosition
EdgeStyle	EdgeWeight
Euclidean	HighlightedEdgeColors
HighlightedEdgeStyle	HighlightedVertexColors
HighlightedVertexStyle	Invariants
LNorm	Large
LoopPosition	LowerLeft
LowerRight	NoMultipleEdges
NoSelfLoops	Normal
NormalDashed	NormalizeVertices
One	Optimum
Parent	PlotRange
RandomInteger	Simple
Small	Strong
Thick	ThickDashed
Thin	ThinDashed
Type	Undirected
UpperLeft	UpperRight
VertexColor	VertexLabel
VertexLabelColor	VertexLabelPosition
VertexNumber	VertexNumberColor
VertexNumberPosition	VertexStyle
VertexWeight	Weak
WeightRange	WeightingFunction
Zoom	

`Combinatorica` options for graph functions.

GetEdgeLabels	GetEdgeWeights
GetVertexLabels	GetVertexWeights
SetEdgeLabels	SetEdgeWeights
SetGraphOptions	SetVertexLabels
SetVertexWeights	

`Combinatorica` functions for graph labels and weights.

AnimateGraph	CircularEmbedding
DilateVertices	GraphOptions
Highlight	RadialEmbedding
RandomVertices	RankGraph
RankedEmbedding	RootedEmbedding
RotateVertices	ShakeGraph
ShowGraph	ShowGraphArray
ShowLabeledGraph	SpringEmbedding
TranslateVertices	Vertices

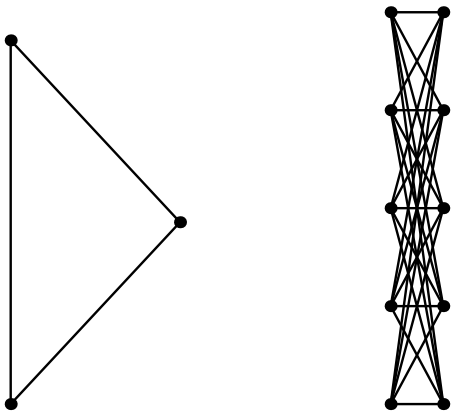
`Combinatorica` functions for drawing graphs.

Generating Graphs

Many graphs consistently prove interesting, in the sense that they are models of important binary relations or have unique graph theoretic properties. Often, these graphs can be parameterized, such as the complete graph on n vertices K_n, giving a concise notation for expressing an infinite class of graphs. Start off with several operations that act on graphs to give different graphs and which, together with parameterized graphs, give the means to construct essentially any interesting graph.

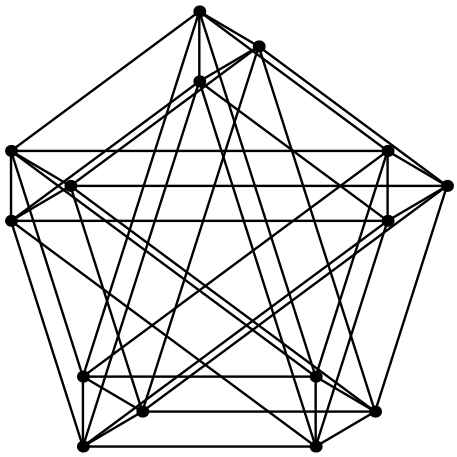
The union of two connected graphs has two connected components.

```
In[42]:= ShowGraph[ GraphUnion[ CompleteGraph[3],
                                CompleteGraph[5,5] ] ];
```



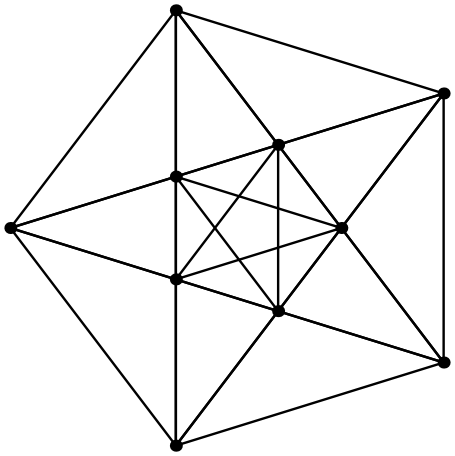
Graph products can be very interesting. The embedding of a product has been designed to show off its structure, and is formed by shrinking the first graph and translating it to the position of each vertex in the second graph.

```
In[43]:= ShowGraph[ GraphProduct[ CompleteGraph[3],
                   CompleteGraph[5] ] ];
```



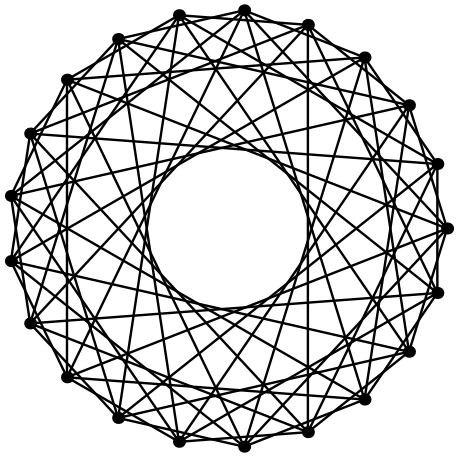
The line graph $L(G)$ of a graph G has a vertex of $L(G)$ associated with each edge of G, and an edge of $L(G)$ if, and only if, the two edges of G share a common vertex.

```
In[44]:= ShowGraph[ LineGraph[CompleteGraph[5]] ];
```



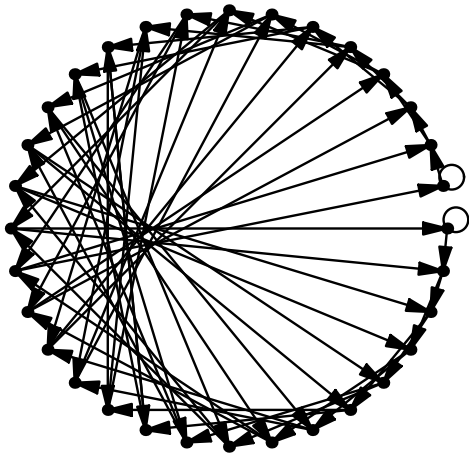
Circulants are graphs whose adjacency matrix can be constructed by rotating a vector n times, and include complete graphs and cycles as special cases. Even random circulant graphs have an interesting, regular structure.

```
In[45]:= ShowGraph[ CirculantGraph[21,
                    RandomKSubset[Range[10],3]]];
```



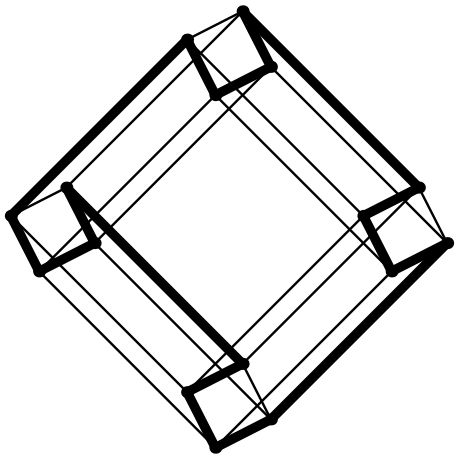
Some graph generators create directed graphs with self-loops, such as this *de Bruijn* or *shift register* graph encoding all length-5 substrings of a binary alphabet.

```
In[46]:= ShowGraph[ DeBruijnGraph[2,5] ];
```



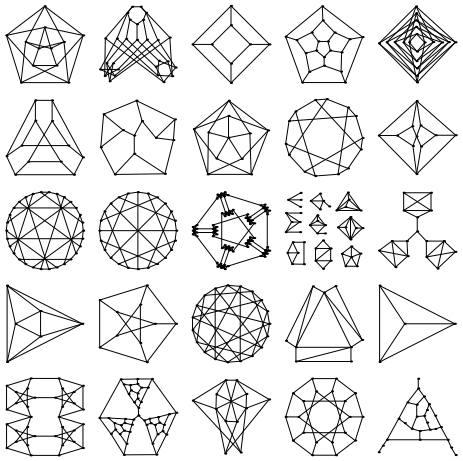
Hypercubes of dimension d are the graph product of cubes of dimension $d-1$ and the complete graph K_2 . Here, a Hamiltonian cycle of the hypercube is highlighted. Colored highlighting and graph animations are also provided in the package.

```
In[47]:= ShowGraph[ Highlight[Hypercube[4],
          {Partition[HamiltonianCycle[Hypercube[4]], 2, 1]}] ];
```



Several of the built-in graph construction functions do not have parameters and only construct a single interesting graph. `FiniteGraphs` collects them together in one list for convenient reference. `ShowGraphArray` permits the display of multiple graphs in one window.

```
In[48]:= ShowGraphArray[Partition[FiniteGraphs,5,5]];
```



BooleanAlgebra	ButterflyGraph
CageGraph	CartesianProduct
ChvatalGraph	CirculantGraph
CodeToLabeledTree	CompleteBinaryTree
CompleteGraph	CompleteKPartiteGraph
CompleteKaryTree	CoxeterGraph
CubeConnectedCycle	CubicalGraph
Cycle	DeBruijnGraph
DodecahedralGraph	EmptyGraph
ExactRandomGraph	ExpandGraph
FiniteGraphs	FolkmanGraph
FranklinGraph	FruchtGraph
FunctionalGraph	GeneralizedPetersenGraph
GraphComplement	GraphDifference
GraphIntersection	GraphJoin
GraphPower	GraphProduct
GraphSum	GraphUnion
GridGraph	GrotztschGraph
Harary	HasseDiagram
HeawoodGraph	HerschelGraph
Hypercube	IcosahedralGraph
IntervalGraph	KnightsTourGraph
LabeledTreeToCode	LeviGraph
LineGraph	ListGraphs
MakeGraph	McGeeGraph
MeredithGraph	MycielskiGraph
NoPerfectMatchingGraph	NonLineGraphs
OctahedralGraph	OddGraph
OrientGraph	Path
PermutationGraph	PetersenGraph
RandomGraph	RandomTree
RealizeDegreeSequence	RegularGraph
RobertsonGraph	ShuffleExchangeGraph
SmallestCyclicGroupGraph	Star
TetrahedralGraph	ThomassenGraph
TransitiveClosure	TransitiveReduction
Turan	TutteGraph
Uniquely3ColorableGraph	UnitransitiveGraph
VertexConnectivityGraph	WaltherGraph
Wheel	

`Combinatorica` functions for graph constructors.

Properties of Graphs

Graph theory is the study of properties or invariants of graphs. Among the properties of interest are such things as connectivity, cycle structure, and chromatic number. Here we demonstrate how to compute several different graph invariants.

An undirected graph is connected if a path exists between any pair of vertices. Deleting an edge from a connected graph can disconnect it. Such an edge is called a bridge.

```
In[49]:= ConnectedQ[ DeleteEdge[ Star[10], {1,10} ] ]

Out[49]= False
```

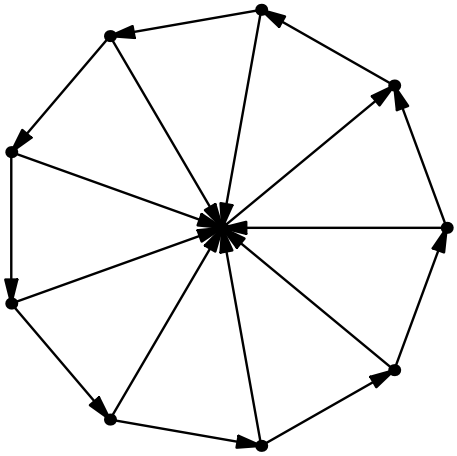
`GraphUnion` can be used to create disconnected graphs.

```
In[50]:= ConnectedComponents[ GraphUnion[CompleteGraph[3],
                CompleteGraph[4]] ]

Out[50]= {{1, 2, 3}, {4, 5, 6, 7}}
```

An orientation of an undirected graph G is an assignment of exactly one direction to each of the edges of G. Note that arrows denoting the direction of each edge are automatically drawn in displaying directed graphs.

```
In[51]:= ShowGraph[ OrientGraph[Wheel[10]] ];
```



An articulation vertex of a graph G is a vertex whose deletion disconnects G. Any graph with no articulation vertices is said to be biconnected. A graph with a vertex of degree 1 cannot be biconnected, since deleting the other vertex that defines its only edge disconnects the graph.

```
In[52]:= BiconnectedComponents[
                RealizeDegreeSequence[{4,4,3,3,3,2,1}] ]

Out[52]= {{2, 7}, {1, 2, 3, 4, 5, 6}}
```

The only articulation vertex of a star is its center, even though its deletion leaves $n-1$ connected components. Deleting a leaf leaves a connected tree.

```
In[53]:= ArticulationVertices[ Star[10] ]

Out[53]= {10}
```

Every edge in a tree is a bridge.

```
In[54]:= Bridges[ RandomTree[10] ]

Out[54]= {{3, 5}, {3, 9}, {7, 9}, {6, 7}, {1, 6}, {2, 8}, {2, 10}, {4, 10}, {1, 10}}
```

A graph is said to be k-connected if there does not exist a set of $k-1$ vertices whose removal disconnects the graph. The wheel is the basic triconnected graph.

```
In[55]:= VertexConnectivity[Wheel[10]]

Out[55]= 3
```

A graph is k-edge-connected if there does not exist a set of $k-1$ edges whose removal disconnects the graph. The edge connectivity of a graph is at most the minimum degree δ, since deleting those edges disconnects the graph. Complete bipartite graphs realize this bound.

```
In[56]:= EdgeConnectivity[CompleteGraph[3,4]]

Out[56]= 3
```

These two complete bipartite graphs are isomorphic, since the order of the two stages is simply reversed. Here, all isomorphisms are returned.

```
In[57]:= Isomorphism[CompleteGraph[3,2], CompleteGraph[2,3], All]

Out[57]= {{3, 4, 5, 1, 2}, {3, 4, 5, 2, 1}, {3, 5, 4, 1, 2}, {3, 5, 4, 2, 1},
          {4, 3, 5, 1, 2}, {4, 3, 5, 2, 1}, {4, 5, 3, 1, 2}, {4, 5, 3, 2, 1},
          {5, 3, 4, 1, 2}, {5, 3, 4, 2, 1}, {5, 4, 3, 1, 2}, {5, 4, 3, 2, 1}}
```

A graph is self-complementary if it is isomorphic to its complement. The smallest nontrivial self-complementary graphs are the path on four vertices and the cycle on five.

```
In[58]:= SelfComplementaryQ[ Cycle[5] ] &&
                SelfComplementaryQ[ Path[4] ]

Out[58]= True
```

A directed graph in which half the possible edges exist is almost certain to contain a cycle. Directed acyclic graphs are often called DAGs.

```
In[59]:= AcyclicQ[
                RandomGraph[100, 0.5, Type -> Directed]
         ]

Out[59]= False
```

The girth of a graph is the length of its shortest cycle. A cage is the smallest possible regular graph (here degree 3) that has a prescribed girth.

```
In[60]:= Girth[ CageGraph[3, 6] ]

Out[60]= 6
```

An Eulerian cycle is a complete tour of all the edges of a graph. An Eulerian cycle of a bipartite graph bounces back and forth between the stages.

```
In[61]:= EulerianCycle[ CompleteGraph[4,4] ]

Out[61]= {7, 2, 8, 1, 5, 4, 6, 3, 7, 4, 8, 3, 5, 2, 6, 1, 7}
```

AcyclicQ	AntiSymmetricQ
BiconnectedQ	BipartiteQ
CliqueQ	CompleteQ
ConnectedQ	EmptyQ
EquivalenceRelationQ	EulerianQ
GraphicQ	HamiltonianQ
IdenticalQ	IndependentSetQ
IsomorphicQ	IsomorphismQ
MultipleEdgesQ	PartialOrderQ
PerfectQ	PlanarQ
PseudographQ	ReflexiveQ
RegularQ	SelfComplementaryQ
SelfLoopsQ	SimpleQ
SymmetricQ	TransitiveQ
TreeIsomorphismQ	TreeQ
TriangleInequalityQ	UndirectedQ
UnweightedQ	VertexCoverQ

`Combinatorica` functions for graph predicates.

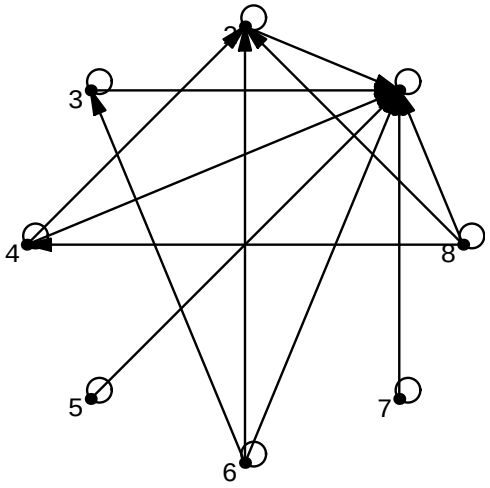
A Hamiltonian cycle of a graph G is a cycle that visits every vertex in G exactly once, as opposed to an Eulerian cycle that visits each edge exactly once. $K_{n,n}$ for $n > 1$ are the only Hamiltonian complete bipartite graphs.

```
In[62]:= HamiltonianCycle[CompleteGraph[3,3], All]

Out[62]= {{1, 4, 2, 5, 3, 6, 1}, {1, 4, 2, 6, 3, 5, 1}, {1, 4, 3, 5, 2, 6, 1}, {1, 4, 3, 6, 2, 5, 1},
          {1, 5, 2, 4, 3, 6, 1}, {1, 5, 2, 6, 3, 4, 1}, {1, 5, 3, 4, 2, 6, 1}, {1, 5, 3, 6, 2, 4, 1},
          {1, 6, 2, 4, 3, 5, 1}, {1, 6, 2, 5, 3, 4, 1}, {1, 6, 3, 4, 2, 5, 1}, {1, 6, 3, 5, 2, 4, 1}}
```

The divisibility relation between integers is reflexive, since each integer divides itself, and anti-symmetric, since x cannot divide y if $x > y$. Finally, it is transitive, as $x \backslash y$ implies $y = c\,x$ for some integer c, so $y \backslash z$ implies $x \backslash z$.

```
In[63]:= ShowGraph[
                g = MakeGraph[Range[8],(Mod[#1,#2]==0)&],
                VertexNumber -> True
         ];
```



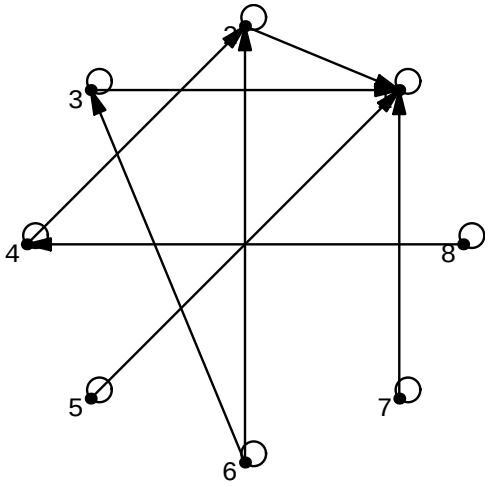
Since the divisibility relation is reflexive, transitive, and anti-symmetric, it is a partial order.

```
In[64]:= PartialOrderQ[g]

Out[64]= True
```

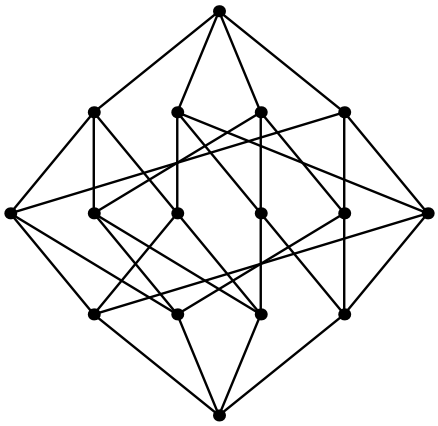
A graph G is transitive if any three vertices x, y, z, such that edges $\{x, y\}$, $\{y, z\} \in G$, imply $\{x, z\} \in G$. The transitive reduction of a graph G is the smallest graph $R(G)$ such that $C(G) = C(R(G))$. The transitive reduction eliminates all implied edges in the divisibility relation, such as $4 \backslash 8$, $1 \backslash 4$, $1 \backslash 6$, and $1 \backslash 8$.

```
In[65]:= ShowGraph[TransitiveReduction[g], VertexNumber -> True];
```



The Hasse diagram clearly shows the lattice structure of the Boolean algebra, the partial order defined by inclusion on the set of subsets.

```
In[66]:= ShowGraph[
           HasseDiagram[MakeGraph[Subsets[4],
           ((Intersection[#2,#1]===#1)&&(#1 != #2))&]]
         ];
```



A topological sort is a permutation p of the vertices of a graph such that an edge $\{i, j\}$ implies i appears before j in p. A complete directed acyclic graph defines a total order, so there is only one possible output from `TopologicalSort`.

```
In[67]:= TopologicalSort[
             MakeGraph[Range[10],(#1 > #2)&] ]

Out[67]= {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

Any labeled graph G can be colored in a certain number of ways with exactly k colors $1, \ldots, k$. This number is determined by the chromatic polynomial of the graph.

```
In[68]:= ChromaticPolynomial[
             GraphUnion[CompleteGraph[2,2], Cycle[3]], z ]

Out[68]= -6 z^2 + 21 z^3 - 29 z^4 + 20 z^5 - 7 z^6 + z^7
```

ArticulationVertices	Automorphisms
BiconnectedComponents	Bridges
ChromaticNumber	ChromaticPolynomial
ConnectedComponents	DegreeSequence
Degrees	DegreesOf2Neighborhood
Diameter	Distances
Eccentricity	EdgeChromaticNumber
EdgeColoring	EdgeConnectivity
Equivalences	EulerianCycle
Girth	GraphCenter
GraphPolynomial	HamiltonianCycle
InDegree	Isomorphism
M	MaximalMatching
MaximumAntichain	MaximumClique
MaximumIndependentSet	MaximumSpanningTree
MinimumChainPartition	MinimumSpanningTree
MinimumVertexColoring	MinimumVertexCover
OutDegree	Radius
Spectrum	StronglyConnectedComponents
TreeToCertificate	V
VertexColoring	VertexConnectivity
VertexCover	WeaklyConnectedComponents

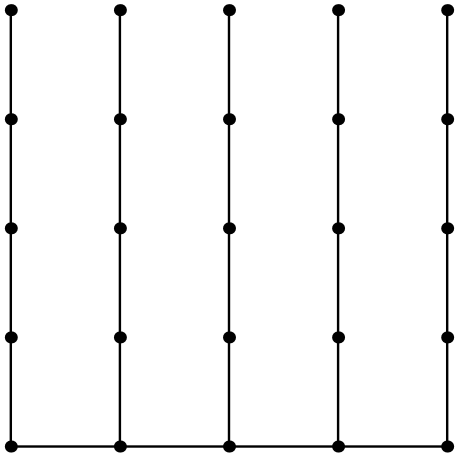
`Combinatorica` functions for graph invariants.

Algorithmic Graph Theory

Finally, there are several invariants of graphs that are of particular interest because of the algorithms that compute them.

The shortest-path spanning tree of a grid graph is defined in terms of Manhattan distance, where the distance between points with coordinates (x, y) and (u, v) is $|x - u| + |y - v|$.

```
In[69]:= ShowGraph[
           ShortestPathSpanningTree[
             GridGraph[5,5],1] ];
```



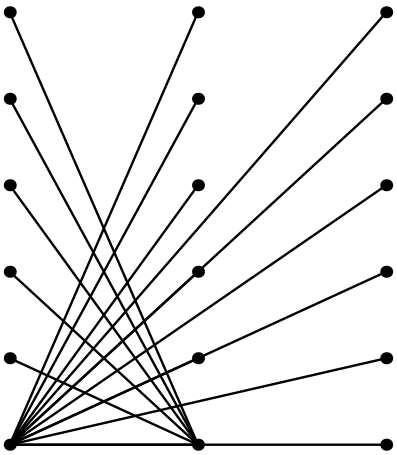
In an unweighted graph, there can be many different shortest paths between any pair of vertices. This path between two opposing corners goes all the way to the right, then all the way to the top.

```
In[70]:= ShortestPath[GridGraph[5,5],1,25]

Out[70]= {1, 2, 3, 4, 5, 10, 15, 20, 25}
```

A minimum spanning tree of a weighted graph is a set of $n - 1$ edges of minimum total weight that form a spanning tree of the graph. Any spanning tree is a minimum spanning tree when the graphs are unweighted.

```
In[71]:= ShowGraph[ MinimumSpanningTree[ CompleteGraph[6,6,6] ] ];
```



The number of spanning trees of a complete graph is n^{n-2}, as was proved by Cayley.

```
In[72]:= NumberOfSpanningTrees[CompleteGraph[10]]

Out[72]= 100000000
```

The maximum flow through an unweighted complete bipartite graph G is the minimum degree $\delta(G)$.

```
In[73]:= NetworkFlow[ CompleteGraph[4,4], 1, 8 ]

Out[73]= 4
```

A matching, in a graph G, is a set of edges of G such that no two of them share a vertex in common. A perfect matching of an even cycle consists of alternating edges in the cycle.

```
In[74]:= BipartiteMatching[ Cycle[8] ]

Out[74]= {{1, 2}, {3, 4}, {5, 6}, {7, 8}}
```

Any maximal matching of a K_n is a maximum matching, and perfect if n is even.

```
In[75]:= MaximalMatching[CompleteGraph[8]]

Out[75]= {{1, 2}, {3, 4}, {5, 6}, {7, 8}}
```

AllPairsShortestPath	AlternatingPaths
ApproximateVertexCover	BellmanFord
BipartiteMatching	BipartiteMatchingAndCover
BreadthFirstTraversal	BrelazColoring
CostOfPath	DepthFirstTraversal
Dijkstra	ExtractCycles
FindCycle	GreedyVertexCover
Neighborhood	NetworkFlow
ParentsToPaths	ResidualFlowGraph
ShortestPath	ShortestPathSpanningTree
StableMarriage	TopologicalSort
TravelingSalesman	TravelingSalesmanBounds
TwoColoring	

`Combinatorica` functions for graph algorithms.