The New Combinatorica

Sriram V. Pemmaraju, The University of Iowa
Steven S. Skiena, State University of New York, Stony Brook
June 15, 2001

Introduction

Combinatorica is a Standard *Mathematica* Add–Onwritten in 1989 by Steven Skiena. It has not been updated since. It has about 230 functions for doing computational discrete mathematics.

Feedback from users, our own expectations of what Combinatorica should be able to do, various advances in graph theory and combinatorics, faster machines, and better versions of *Mathematica* are factors that have motivated this rewrite. About 80% of the functions have been rewritten and the package now contains about 350 functions.

The new Combinatorica provides functions for enumerating, selecting, ranking, and unranking various combinatorial objects such as permutations, combinations, integer partitions, set partitions, Young tableaux, trees, and graphs. It also provides functions to generate various classes of graphs and provides functions for all the standard graph algorithms. The specific ways in which the new Combinatorica improves over the old version are as follows:

–Improved graph data structure, especially tuned for sparse graphs

–Functions provided for additional topics such as Set Partitions and Polya Theory

–Better graphics, with graph drawing significantly improved

–Many functions substantially speeded up

–Many old functions now have additonal functionality providing users greater flexibility, ease of use, and more error checking

–New graph instances and graph classes

–Many miscellaneous new functions

Each of these items is examined below in some detail.

Old Combinatorica vs New Combinatorica

Better Graph Data Structure

A graph is now represented as a triple, the first element is an edge list, the second element is the embedding of the vertices, and the third element is optional graphics information. The main difference is that the adjacency matrix representation has been replaced by an edge list representation. The implications of this change are felt throughout the package –in running time improvements, memory savings, increased functionality, and better graph drawings. The package can now work with graphs that are about 50 times larger than graphs that Old Combinatorica could deal with.

```
g = CompleteGraph[4]
```

```
-Graph:<6, 4, Undirected>-
```

```
g[[1]]
```

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

```
g[[2]]
```

```
{{{0, 1.}}, {{-1., 0}}, {{0, -1.}}, {{1., 0}}}
```

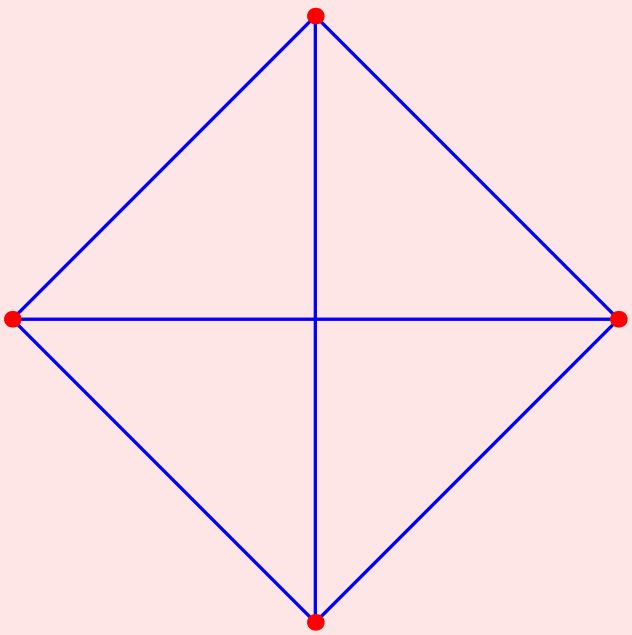
```
g[[0]]
```

```
Graph
```

```
g = SetGraphOptions[g, VertexColor -> Red, EdgeColor -> Blue]
```

```
-Graph:<6, 4, Undirected>-
```

```
ShowGraph[g]
```

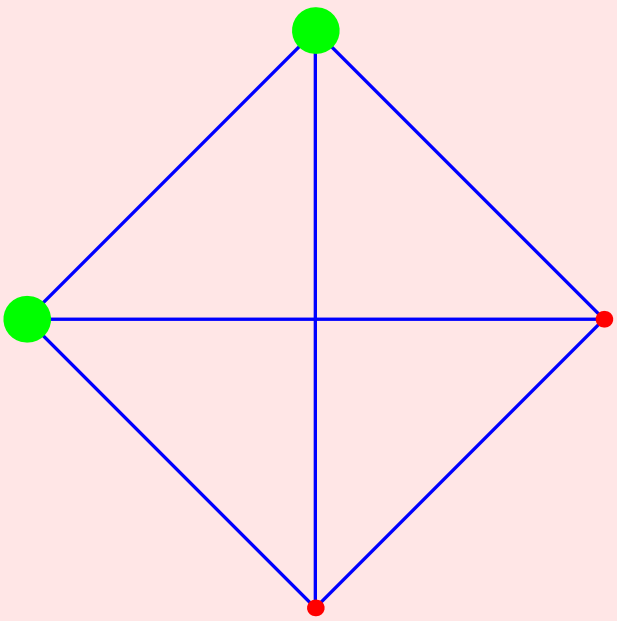


```
- Graphics -
```

```
g = SetGraphOptions[g,
  {{{1, 2}, VertexStyle -> Disc[Large], VertexColor -> Green}}]
```

```
-Graph:<6, 4, Undirected>-
```

```
ShowGraph[g, PlotRange → Large[0.05]]
```



```
- Graphics -
```

```
Length[g]
```

```
4
```

```
g[[1]]
```

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

```
g[[2]]
```

```
{{{0, 1.}, VertexStyle → Disc[Large], VertexColor → RGBColor[0., 1., 0.]},
 {{-1., 0}, VertexStyle → Disc[Large], VertexColor → RGBColor[0., 1., 0.]},
 {{0, -1.}}, {{1., 0}}}
```

```
g[[3]]
```

```
VertexColor → RGBColor[1., 0., 0.]
```

```
g[[4]]
```

```
EdgeColor → RGBColor[0., 0., 1.]
```

This data structure allows us to store graphics information that pertains to the entire graph or to individual elements such as edges and vertices.

For sparse graphs the savings in memory is dramatic. For dense graphs there is no significant difference in memory usage.

```
g = DiscreteMath`OldCombinatorica`GridGraph[20, 20];
```

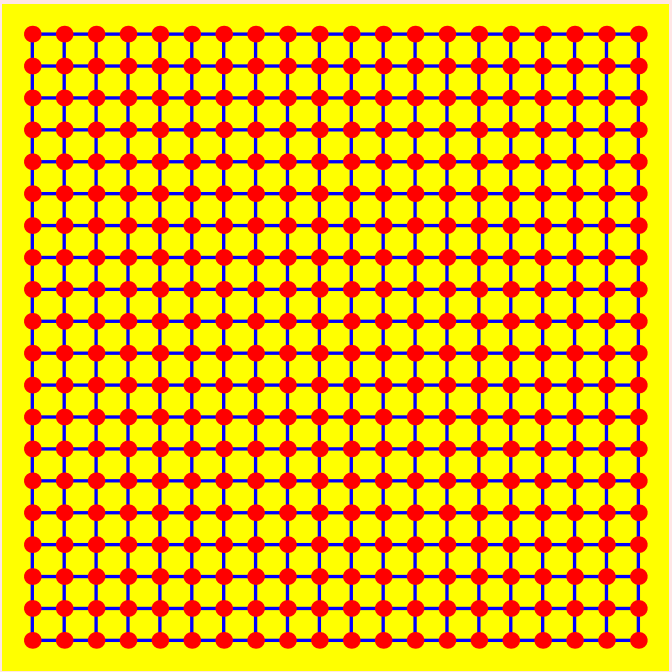
```
ByteCount[g]
```

```
2593664
```

```
g = GridGraph[20, 20]; ByteCount[g]
```

```
61224
```

```
h = SetGraphOptions[g, VertexColor → Red, EdgeColor → Blue];
ShowGraph[h, Background → Yellow]
```



```
- Graphics -
```

```
ByteCount[h]
```

```
61432
```

```
g = DiscreteMath`OldCombinatorica`RandomGraph[100, .5]; ByteCount[g]
```

```
168448
```

```
g = RandomGraph[100, .5]; ByteCount[g]
```

```
219812
```

Additional topics

A number of functions relating to **Polya theory** and **set partitions** have been added to the new Combinatorica.

■ Polya Theory

Groups commonly used in Polya theory can now be generated.

```
?DihedralGroup
```

```
DihedralGroup[n] returns the dihedral group
  on n symbols. Note that the order of this group is 2n.
```

```
DihedralGroup[4]
```

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

The cycle index of groups commonly used in Polya theory can now be computed.

```
?SymmetricGroupIndex
```

```
SymmetricGroupIndex[n, x] returns the cycle index of the symmetric group
  on n symbols, expressed as a polynomial in x[1], x[2], ..., x[n].
```

```
SymmetricGroupIndex[6, x]
```

$$\frac{x[1]^6}{720} + \frac{1}{48} x[1]^4 x[2] + \frac{1}{16} x[1]^2 x[2]^2 + \frac{x[2]^3}{48} + \frac{1}{18} x[1]^3 x[3] + \frac{1}{6} x[1] x[2] x[3] + \frac{x[3]^2}{18} + \frac{1}{8} x[1]^2 x[4] + \frac{1}{8} x[2] x[4] + \frac{1}{5} x[1] x[5] + \frac{x[6]}{6}$$

```
?DihedralGroupIndex
```

```
DihedralGroupIndex[n, x] returns the cycle index of the dihedral group
  on n symbols, expressed as a polynomial in x[1], x[2], ...,x[n].
```

```
DihedralGroupIndex[10, y]
```

$$\frac{y[1]^{10}}{20} + \frac{1}{4} y[1]^2 y[2]^4 + \frac{3 y[2]^5}{10} + \frac{y[5]^2}{5} + \frac{y[10]}{5}$$

Non−isomorphicinstances of various combinatorial objects can now be counted and enumerated. First we show examples involving graphs.

```
?GraphPolynomial
```

```
GraphPolynomial[n, x] returns a polynomial in x in which the
  coefficient of x^m is the number of non-isomorphic graphs with
  n vertices and m edges. GraphPolynomial[n, x, Directed] returns
  a polynomial in x in which the coefficient of x^m is the number
  of non-isomorphic directed graphs with n vertices and m edges
```

```
GraphPolynomial[6, x]
```

$$1 + x + 2 x^2 + 5 x^3 + 9 x^4 + 15 x^5 + 21 x^6 + 24 x^7 + 24 x^8 + 21 x^9 + 15 x^{10} + 9 x^{11} + 5 x^{12} + 2 x^{13} + x^{14} + x^{15}$$

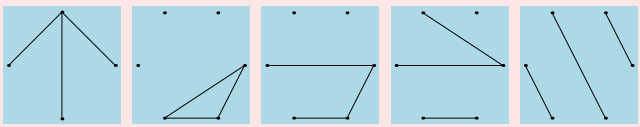
```
?NumberOfGraphs
```

```
NumberOfGraphs[n] returns the number of non-isomorphic undirected
  graphs with n vertices. NumberOfGraphs[n, m] returns the number
  of non-isomorphic undirected graphs with n vertices and m edges.
```

```
Table[NumberOfGraphs[i], {i, 1, 15}] // TableForm
```

```
1
2
4
11
34
156
1044
12346
274668
12005168
1018997864
165091172592
50502031367952
29054155657235488
31426485969804308768
```

```
ShowGraphArray[ListGraphs[6, 3], Background -> LightBlue]
```



```
- GraphicsArray -
```

Now we show examples involving "necklaces." In the first example below, we show all the distinct necklaces with 5 beads, colored red and blue.

```
?ListNecklaces
```

```
ListNecklaces[n, c, Cyclic] returns all distinct necklaces
  whose beads are colored by colors from c. Here c is a
  list of n, not necessarily distinct colors and two colored
  necklaces are considered equivalent if one can be obtained
  by rotating the other. ListNecklaces[n, c, Dihedral] is
  similar except that two necklaces are considered equivalent
  if one can be obtained from the other by a rotation or a flip
```

```
ListNecklaces[5, {r, r, b, b, y}, Dihedral]
```

```
{{r, r, b, b, y}, {b, b, r, y, r}, {b, r, b, r, y}, {b, r, r, b, y}}
```

```
ListNecklaces[5, {r, r, b, b, y}, Cyclic]
```

```
{{r, r, b, b, y}, {b, b, r, r, y}, {b, b, r, y, r},
 {b, r, b, r, y}, {b, r, b, y, r}, {b, r, r, b, y}}
```

```
?NumberOfNecklaces
```

```
NumberOfNecklaces[n, nc, Cyclic] returns the number of distinct ways in
  which an n-bead necklace can be colored with nc colors assuming
  that two colorings are equivalent if one can be obtained from the
  other by a rotation. NumberOfNecklaces[n, nc, Dihedral] returns
  the number of distinct ways in which an n-bead necklace can be
  colored with nc colors assuming that two colorings are equivalent
  if one can be obtained from the other by a rotation or a flip
```

```
Table[NumberOfNecklaces[i, 2, Dihedral], {i, 10, 20}] // TableForm
```

```
78
126
224
380
687
1224
2250
4112
7685
14310
27012
```

```
?NecklacePolynomial
```

```
NecklacePolynomial[n, c, Cyclic] returns a polynomial in the colors
  in c whose coefficients represent number of ways of coloring
  an n-bead necklace with colors chosen from c assuming that
  two colorings are equivalent if one can be obtained from the
  other by a rotation. NecklacePolynomial[n, c, Dihedral] is
  different in that it considers two colorings equivalent if one
  can be obtained from the other by a rotation or a flip or both.
```

```
NecklacePolynomial[10, {R, B, G}, Cyclic]
```

$$B^{10} + B^9 G + 5 B^8 G^2 + 12 B^7 G^3 + 22 B^6 G^4 + 26 B^5 G^5 + 22 B^4 G^6 + 12 B^3 G^7 + 5 B^2 G^8 + B G^9 + G^{10} + B^9 R + 9 B^8 G R + 36 B^7 G^2 R + 84 B^6 G^3 R + 126 B^5 G^4 R + 126 B^4 G^5 R + 84 B^3 G^6 R + 36 B^2 G^7 R + 9 B G^8 R + G^9 R + 5 B^8 R^2 + 36 B^7 G R^2 + 128 B^6 G^2 R^2 + 252 B^5 G^3 R^2 + 318 B^4 G^4 R^2 + 252 B^3 G^5 R^2 + 128 B^2 G^6 R^2 + 36 B G^7 R^2 + 5 G^8 R^2 + 12 B^7 R^3 + 84 B^6 G R^3 + 252 B^5 G^2 R^3 + 420 B^4 G^3 R^3 + 420 B^3 G^4 R^3 + 252 B^2 G^5 R^3 + 84 B G^6 R^3 + 12 G^7 R^3 + 22 B^6 R^4 + 126 B^5 G R^4 + 318 B^4 G^2 R^4 + 420 B^3 G^3 R^4 + 318 B^2 G^4 R^4 + 126 B G^5 R^4 + 22 G^6 R^4 + 26 B^5 R^5 + 126 B^4 G R^5 + 252 B^3 G^2 R^5 + 252 B^2 G^3 R^5 + 126 B G^4 R^5 + 26 G^5 R^5 + 22 B^4 R^6 + 84 B^3 G R^6 + 128 B^2 G^2 R^6 + 84 B G^3 R^6 + 22 G^4 R^6 + 12 B^3 R^7 + 36 B^2 G R^7 + 36 B G^2 R^7 + 12 G^3 R^7 + 5 B^2 R^8 + 9 B G R^8 + 5 G^2 R^8 + B R^9 + G R^9 + R^{10}$$

■ Set Partitions

```
?SetPartitions
```

```
SetPartitions[set] returns the list of set partitions of set.
  SetPartitions[n] returns the list of set partitions of {1, 2,..., n}.
```

SetPartitions[4]

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

?KSetPartitions

```
KSetPartitions[set, k] returns the list of set partitions
  of set with k blocks. KSetPartitions[n, k] returns the
  list of set of partitions of {1, 2, ..., n} with k blocks.
```

KSetPartitions[5, 3]

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

?RankSetPartition

```
RankSetPartition[sp, s] ranks sp in the list of all set partitions
  of set s. RankSetPartition[sp] ranks sp in the list of all set
  partitions of the set of elements that appear in any subset in sp.
```

RankSetPartition[{{1, 2}, {3, 5}, {4}}]

```
29
```

?UnrankSetPartition

```
UnrankSetPartition[r, set] finds a set partition of set with rank r.
  UnrankSetPartition[r, n] finds a set partition of [n] with rank r.
```

UnrankSetPartition[%, 5]

```
UnrankSetPartition[Null, 5]
```

```
?RandomSetPartition
```

```
RandomSetPartition[set] returns a random
  set partition of set. RandomSetPartition[n] returns
  a random set partition of the first n natural numbers.
```

```
RandomSetPartition[10]
```

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

```
?RandomKSetPartition
```

```
RandomKSetPartition[set, k] returns a random set partition of set
  with k blocks.  RandomKSetPartition[n, k] returns a random
  set partition of the first n natural numbers into k blocks.
```

```
RandomKSetPartition[10, 4]
```

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

```
?BellB
```

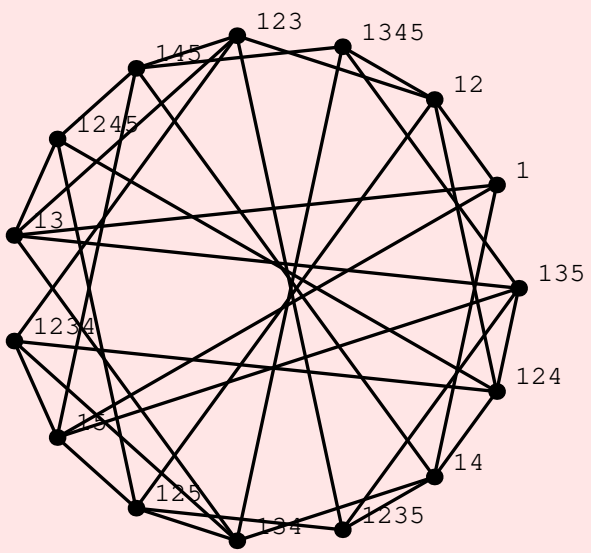
```
BellB[n] returns the nth Bell number
```

```
BellB[300]
```

```
9593717160839270277309012597458244643669761253486524090465101452308505449075
 5557949670979914220944478133617034617035274839234529106001070942419778835524
 9953791425694031091482644794939518996181309914949469240123116264668354144469
 8052769000667336121756179876704099764167712726433111430458732053150116078018
 8246258278658246389449826531609243182040031829104894020820811280384631732800
 1600124901176597068501042030359075102729529486736608734055663641171003800990
 645
```

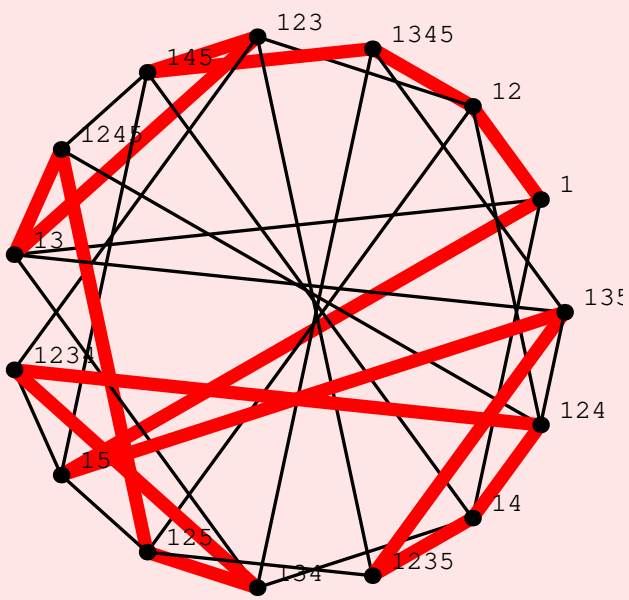
The following graph g shows 2−blockset partitions of {1, 2, 3, 4, 5} connected by edges whenever a set partition can be obtained from another by deleting an element from a block and inserting it elsewhere. In the graph below that we show a Hamiltonian cycle in g indicating a "gray code" ordering of the 2−blockset partitions.

```
l52 = Map[StringJoin[Map[ToString, #[[1]]]] &, sp52 = KSetPartitions[5, 2]];
ShowGraph[g = SetVertexLabels[MakeGraph[sp52, (MemberQ[{1, 4},
       Sum[Abs[Position[#1, i][[1, 1]] - Position[#2, i][[1, 1]]], {i, 5}]]) &,
    Type -> Undirected], l52], PlotRange → Large[0.2]]
```




```
- Graphics -
```

```
ShowGraph[Highlight[g, {Partition[HamiltonianCycle[g], 2, 1]}, {Red}],
 PlotRange → Large[0.1]]
```




```
- Graphics -
```

The graph above has not 1, but more than 6000 Hamiltonian cycles!

```
Length[HamiltonianCycle[g, All]]
```

```
6528
```

Our implementation of the Stirling number of the second kind is not recursive. It uses an identity that expresses these Stirling numbers as the signed sum of binomial numbers. This makes our implementation faster than the *Mathematica* implementation.

```
StirlingSecond[100, 50]
```

```
43098323700936634042151430154725869594352028961434061391244174113128031905885378314559826165999201390 0
```

```
Table[Timing[StirlingS2[100 i, 50 i];] /
  Timing[StirlingSecond[100 i, 50 i];], {i, 2, 7}]
```

```
{{4., 1}, {3., 1}, {2.5, 1}, {2.66667, 1}, {3.11111, 1}, {2.85714, 1}}
```

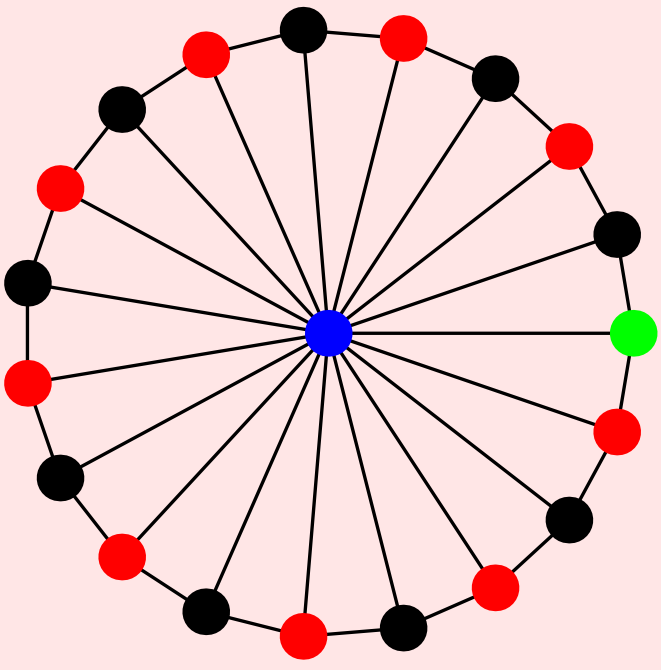
Better Graphics

Old Combinatorica cannot display the colored graph. On the other hard, consider what-New Combinatorica can do.

```
c = VertexColoring[ g = Wheel[20]]
```

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

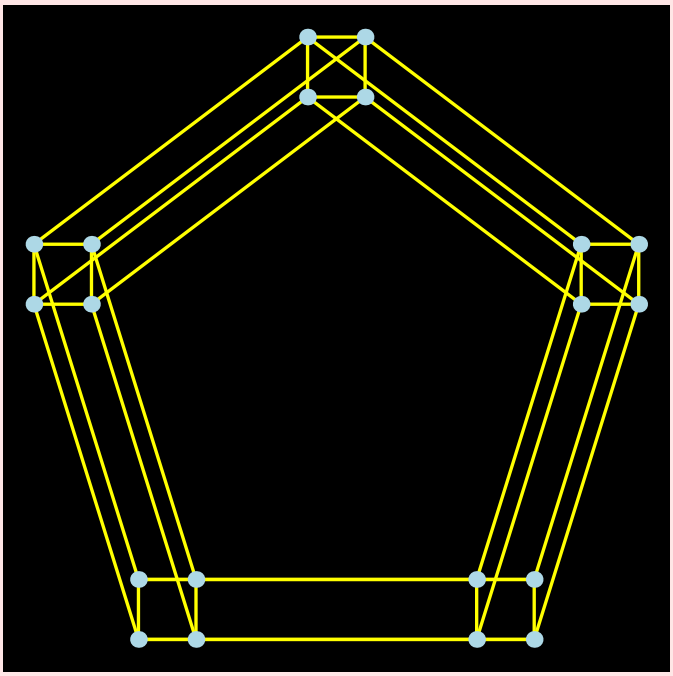
```
ShowGraph[Highlight[ g, Table[ Flatten[Position[c, i]], {i, Max[c]}]]]
```



```
- Graphics -
```

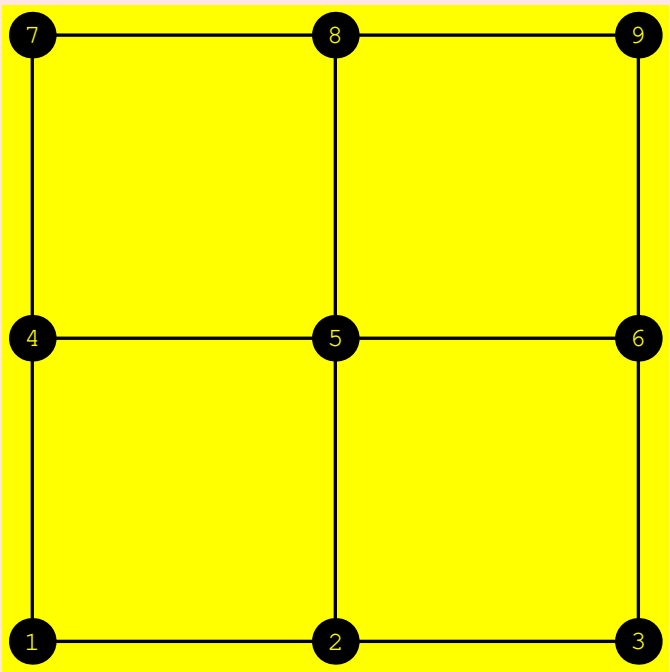
Here are some pictures we can easily create in the New Combinatorica.

```
g = GraphProduct[GridGraph[2, 2], Cycle[5]];
ShowGraph[g, VertexColor -> LightBlue,
 EdgeColor -> Yellow, Background -> Black]
```



```
- Graphics -
```

```
g = GridGraph[3, 3];
ShowGraph[g, VertexStyle -> Disc[Large], VertexNumber -> Center,
 VertexNumberColor -> Yellow, Background -> Yellow]
```

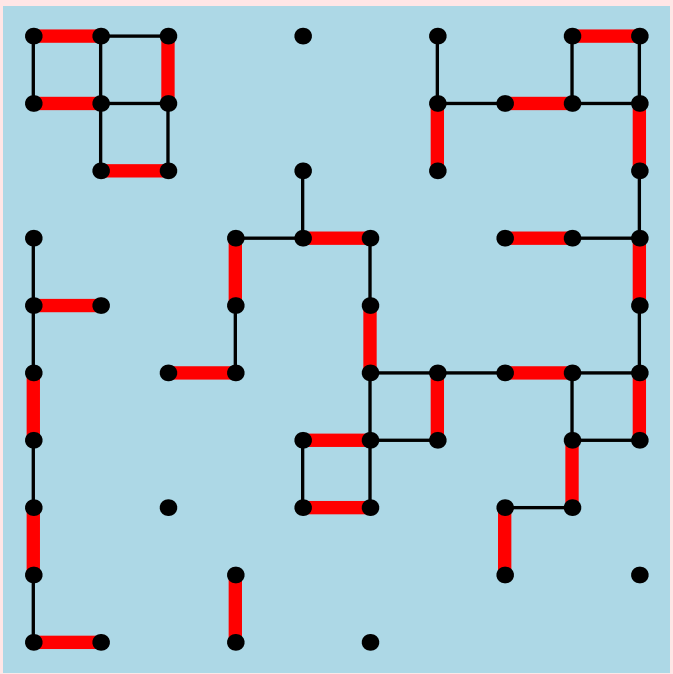



```
- Graphics -
```

```
g = InduceSubgraph[ GridGraph[10, 10], RandomSubset[ Range[100]]];
m = BipartiteMatching[ g ]
```

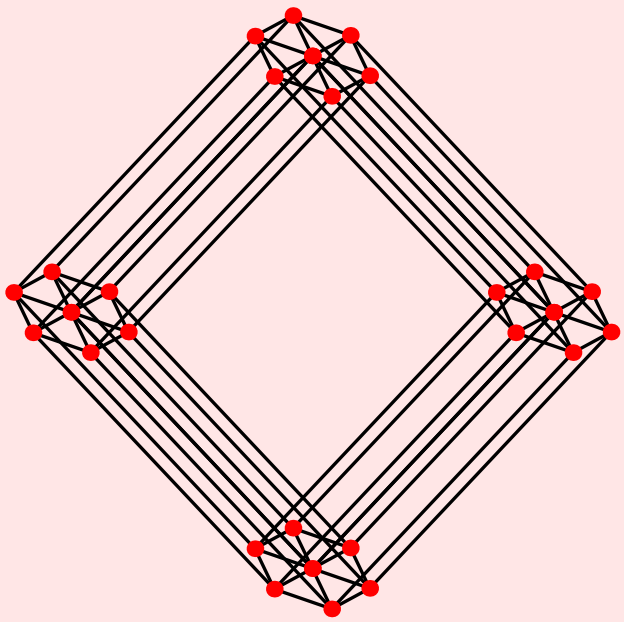
```
{{1, 2}, {3, 6}, {5, 9}, {7, 13}, {11, 12}, {14, 19}, {15, 21},
 {16, 17}, {18, 25}, {20, 28}, {22, 23}, {24, 32}, {26, 27},
 {29, 30}, {31, 35}, {33, 40}, {36, 37}, {38, 39}, {41, 42},
 {44, 49}, {45, 52}, {46, 47}, {48, 55}, {50, 51}, {53, 54}, {58, 59}}
```

```
ShowGraph[Highlight[g, {m}, {Red}], Background -> LightBlue]
```



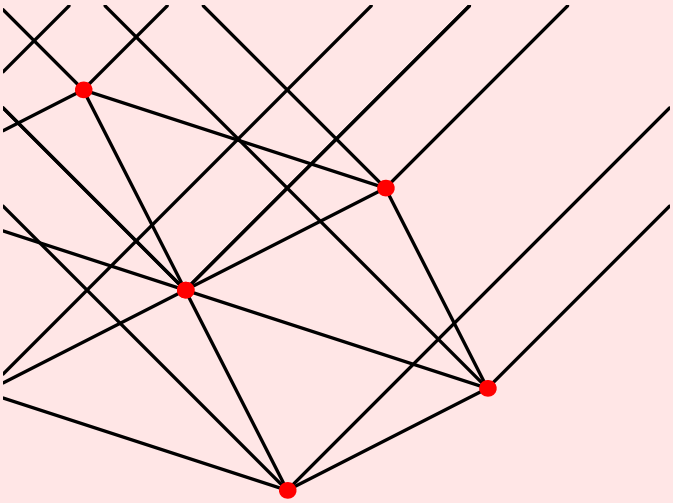
```
- Graphics -
```

```
ShowGraph[g = Hypercube[5], VertexColor -> Red]
```



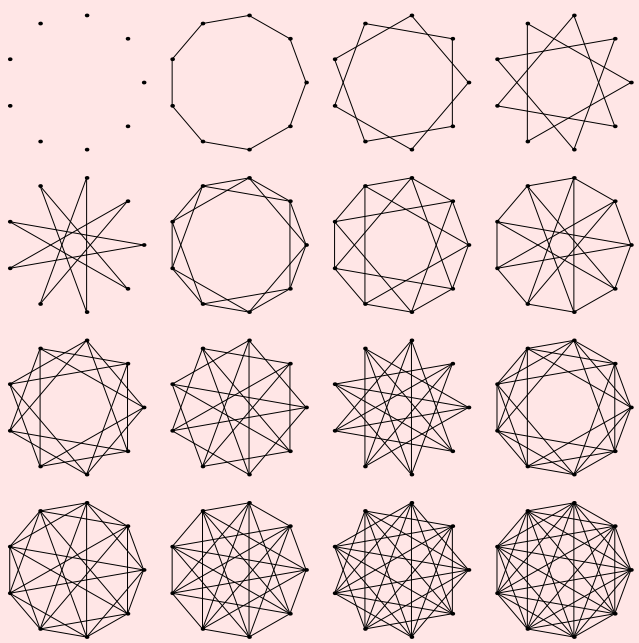
```
- Graphics -
```

```
ShowGraph[g, VertexColor -> Red, PlotRange -> Zoom[{1, 2, 3, 4}]]
```



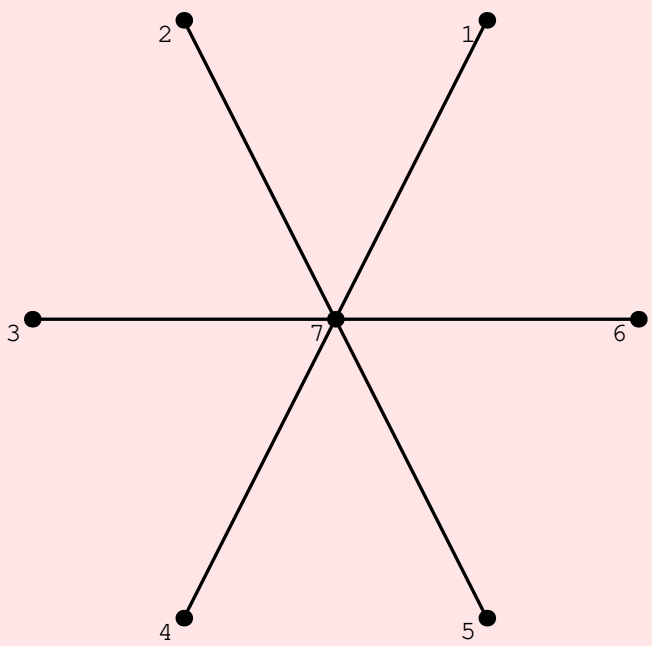
```
- Graphics -
```

```
ShowGraphArray[
 Partition[l = Union[Map[(CirculantGraph[9, #]) &, Subsets[8]]], 4, 4]]
```



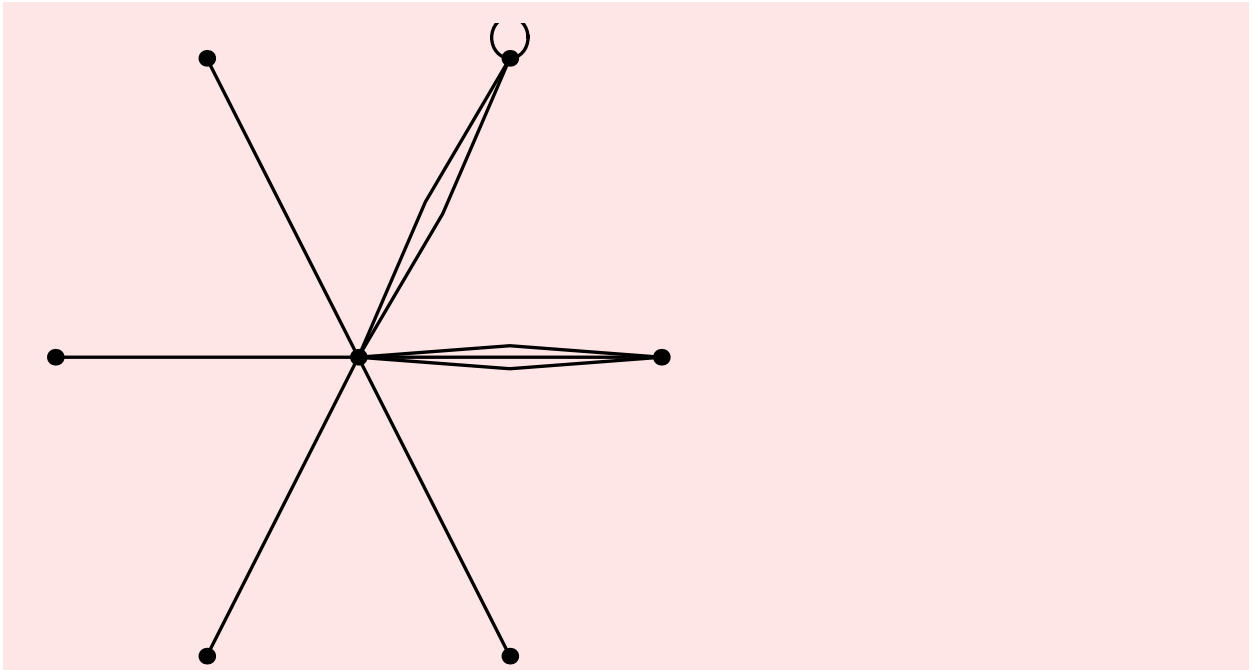
- GraphicsArray -

```
g = Star[7]; ShowGraph[g, VertexNumber → On]
```




```
- Graphics -
```

```
ShowGraph[AddEdges[g, {{1, 7}, {1, 1}, {6, 7}, {7, 6}}]]
```



```
- Graphics -
```

Improvement in Run Times

Many functions in the Old Combinatorica are quite slow. This is for a variety of reasons. Some of the reasons cannot be remedied. For example, the overhead of using *Mathematica* or the inherent intractability of certain problems. However, using better algorithms, better data structures, better programming practices, and new *Mathematica* features we have speeded up most functions, some by several orders of magnitude.

The speedup you see in the following examples is because the adjacency matrix representation for graphs has been replaced by an edge list representation. This is ideal for the sparse graphs being generated below.

```
{Timing[ DiscreteMath`OldCombinatorica`Path[300];], Timing[Path[300];]}
```

```
{{466.35 Second, Null}, {0. Second, Null}}
```

```
{Timing[ DiscreteMath`OldCombinatorica`RandomTree[200];],
 Timing[RandomTree[200];]}
```

```
{{1.01 Second, Null}, {0.23 Second, Null}}
```

```
{Timing[ DiscreteMath`OldCombinatorica`GridGraph[30, 30];],
 Timing[GridGraph[30, 30];]}
```

```
{{0.72 Second, Null}, {0.12 Second, Null}}
```

Various utility functions that convert between different representations of graphs have also been speeded up and this has led to speedup of almost all the graph algorithms in the package.

Observe the remarkable speedup in the function ToAdjacencyLists, shown below. ToAdjacencyLists is used in most graph algorithms. Below we attempt to show the difference in the running times of MinimumSpanningTree in Old Combinatorica and New Combinatorica. The attempt is unsuccessful because MinimumSpanningTree in Old Combinatorica takes more time than we could spend waiting! Some of this difference in running time is due to the difference in the running times of ToAdjacencyLists.

```
g = DiscreteMath`OldCombinatorica`GridGraph[30, 30];
Timing[ DiscreteMath`OldCombinatorica`ToAdjacencyLists[g];]
```

```
{6.23 Second, Null}
```

```
g = GridGraph[30, 30]; Timing[ ToAdjacencyLists[g];]
```

```
{0.1 Second, Null}
```

```
g = SetEdgeWeights[GridGraph[30, 30]]; Timing[ MinimumSpanningTree[g];]
```

```
{1.05 Second, Null}
```

```
g =  DiscreteMath`OldCombinatorica`GridGraph[20, 20];
Timing[DiscreteMath`OldCombinatorica`MinimumSpanningTree[g];]
```

```
$Aborted
```

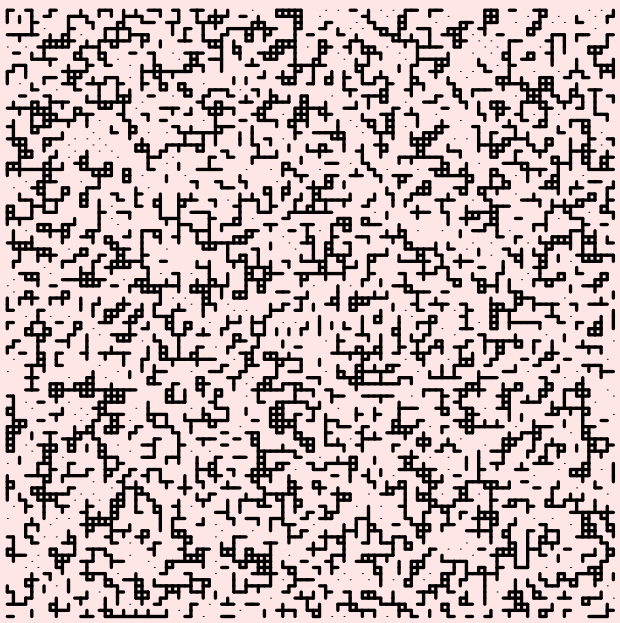
We can now compute with fairly large size sparse graphs. Here we construct a random subgraph with about 5000 vertices and edges of a graph with about 10,000 vertices and 40,000 edges. The subgraph has 668 connected components and it took about 6 seconds to compute these.

```
g = GridGraph[100, 100]; h = InduceSubgraph[g, RandomSubset[Range[10000]]];
```

```
{V[h], M[h]}
```

```
{4987, 4979}
```

```
ShowGraph[h, VertexStyle -> Disc[0]]
```



```
- Graphics -
```

```
Timing[c = ConnectedComponents[h];]
```

```
{2.41 Second, Null}
```

```
Length[c]
```

```
672
```

Some speedup is achieved by an improvement in implementation –more careful attention is paid to ensuring that the actual running time of the implementation is the same as what is promised by theory. The examples below show speedups achieved in functions such as LineGraph and VertexColoring.

```
g = DiscreteMath`OldCombinatorica`RandomGraph[100, .3];
Timing[DiscreteMath`OldCombinatorica`LineGraph[g];]
```

```
{10.73 Second, Null}
```

```
g = RandomGraph[100, .3]; Timing[LineGraph[g];]
```

```
{0.99 Second, Null}
```

```
g = DiscreteMath`OldCombinatorica`Wheel[200];
Timing[DiscreteMath`OldCombinatorica`VertexColoring[ g ];]
```

```
{3.4 Second, Null}
```

```
g = Wheel[200]; Timing[ VertexColoring[g]; ]
```

```
{0.09 Second, Null}
```

Speedup in some cases was achieved by compiling carefully selected functions. In the example below, LexicographicPermutations is reimplemented as an iterative (rather than as a recursive function) and then compiled.

```
Timing[DiscreteMath`OldCombinatorica`LexicographicPermutations[Range[8]];]
```

```
{4.21 Second, Null}
```

```
Timing[LexicographicPermutations[8];]
```

```
{0.36 Second, Null}
```

In some implementations, we took advantage of the new packed array implementation of lists. Here is a comparison between our new implementation of ToCycles and the old implementation of ToCycles from the Permutations package.

```
p = DiscreteMath`Permutations`RandomPermutation[1000];
Timing[DiscreteMath`Permutations`ToCycles[p];]
```

```
{5.36 Second, Null}
```

```
p = RandomPermutation[1000]; Timing[ToCycles[p];]
```

```
{0.07 Second, Null}
```

In fact, ToCyles is so fast that the function SignaturePermutation that computes the sign of a permutation is significantly faster than the corresponding *Mathematica* function Signature.

```
p = RandomPermutation[5000];
{Timing[Signature[p];], Timing[SignaturePermutation[p];]}
```

```
{{4.02 Second, Null}, {0.25 Second, Null}}
```

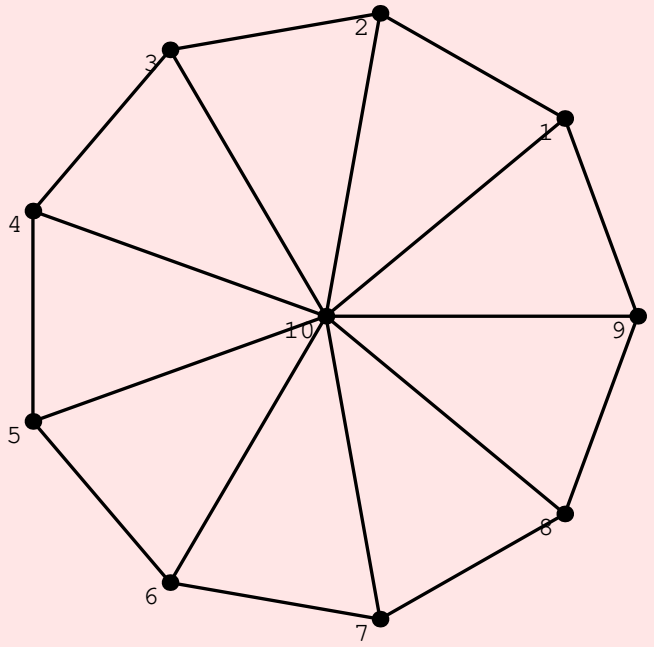
Better Functionality

A variety of old functions are now easier to use. A variety of new user friendly functions have been added, Options have been used to make several of the functions more user friendly and flexible. For example, AddEdges allows many variations of edge specifications.

```
?AddEdges
```

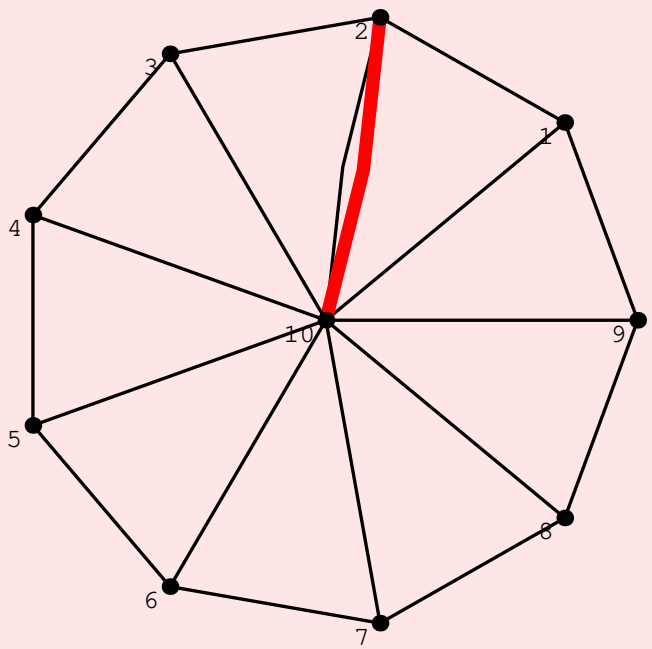
```
AddEdges[g, edgeList] gives graph g with the new edges in
  edgeList added. edgeList can have the form {a, b} if we want
  to add a single edge {a, b} or the form {{a, b}, {c, d}, ...},
  if we want to add edges {a, b}, {c, d}, ... or the form
  { {{a, b}, x}, {{c, d}, y}, ...} where x and y are graphics
  information associated with {a, b} and {c, d} respectively.
```

```
g = Wheel[10]; ShowGraph[g, VertexNumber → On]
```



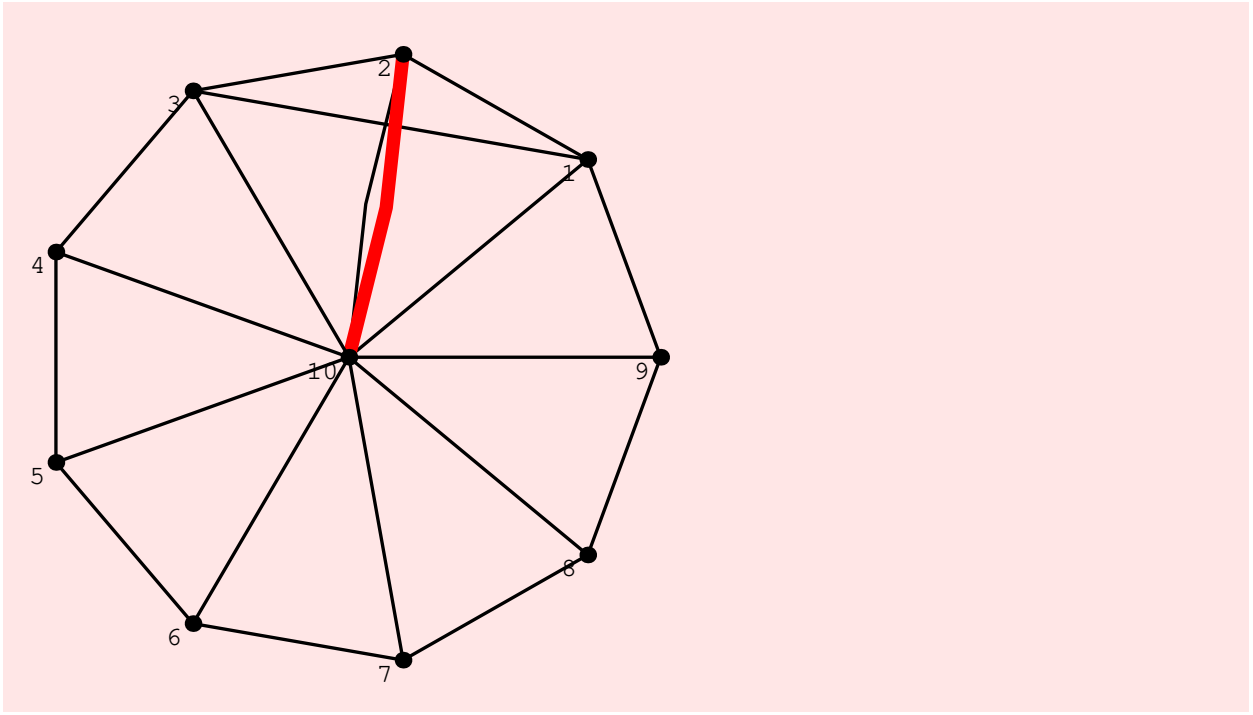

```
- Graphics -
```

```
ShowGraph[g = AddEdges[g, {{{2, 10}, EdgeColor → Red, EdgeStyle → Fat}}],
 VertexNumber → On]
```




```
- Graphics -
```

```
ShowGraph[g = AddEdges[g, {3, 1}], VertexNumber → On]
```




```
- Graphics -
```

Here is another example, in which we show how the function BreadthFirstTraversal has become much more flexible. First compare the usage messages for the new BreadthFirstTraversal and the old BreadthFirstTraversal. Some additional tags allow the user to get different kinds of information from BreadthFirstTraversal.

```
?BreadthFirstTraversal
```

```
BreadthFirstTraversal[g,v] performs a breadth-first traversal
  of graph g starting from vertex v, and gives the breadth-
  first numbers of the vertices. BreadthFirstTraversal[g,v,
  Edge] returns the edges of the graph that are traversed by
  breadth-first traversal. BreadthFirstTraversal[g,v,Tree]
  returns the breadth-first search tree. BreadthFirstTraversal[
  g,v,Level] returns the level number of the vertices.
```

```
?DiscreteMath`OldCombinatorica`BreadthFirstTraversal
```

```
BreadthFirstTraversal[g,v] performs a breadth-first
  traversal of graph g starting from vertex v, and gives a
  list of vertices in the order in which they were encountered.
```

```
g = GridGraph[5, 5];
```

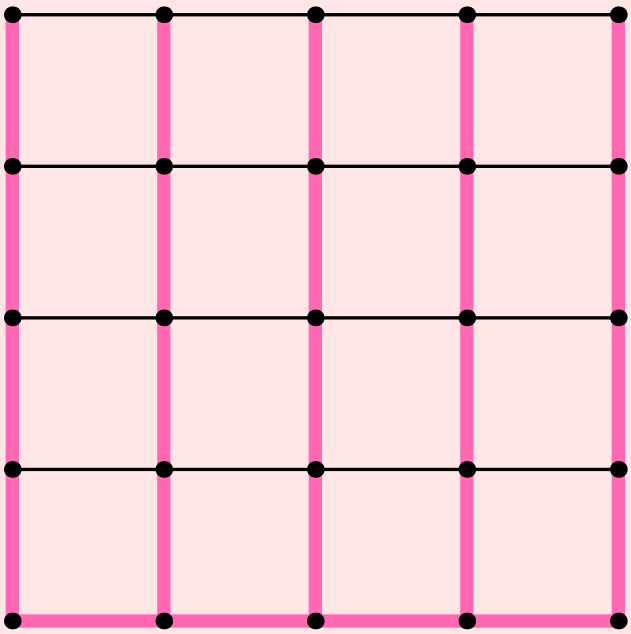
```
BreadthFirstTraversal[g, 1]
```

```
{1, 2, 4, 7, 11, 3, 5, 8, 12, 16, 6, 9,
 13, 17, 20, 10, 14, 18, 21, 23, 15, 19, 22, 24, 25}
```

```
e = BreadthFirstTraversal[g, 1, Edge]
```

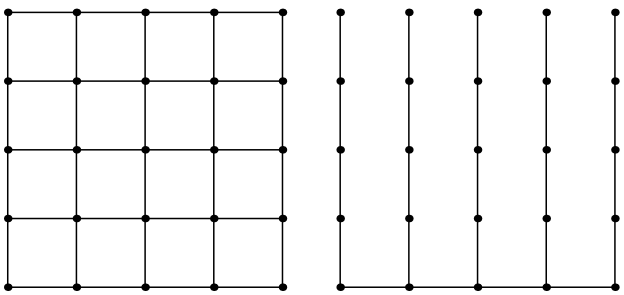
```
{{1, 2}, {2, 3}, {3, 4}, {4, 5}, {1, 6}, {2, 7}, {3, 8}, {4, 9}, {5, 10},
 {6, 11}, {7, 12}, {8, 13}, {9, 14}, {10, 15}, {11, 16}, {12, 17}, {13, 18},
 {14, 19}, {15, 20}, {16, 21}, {17, 22}, {18, 23}, {19, 24}, {20, 25}}
```

```
ShowGraph[Highlight[g, {e}, {HotPink}]]
```



```
- Graphics -
```

```
ShowGraphArray[{g, BreadthFirstTraversal[g, 1, Tree]}]
```



```
- GraphicsArray -
```

```
BreadthFirstTraversal[g, 1, Level]
```

```
{0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7, 4, 5, 6, 7, 8}
```

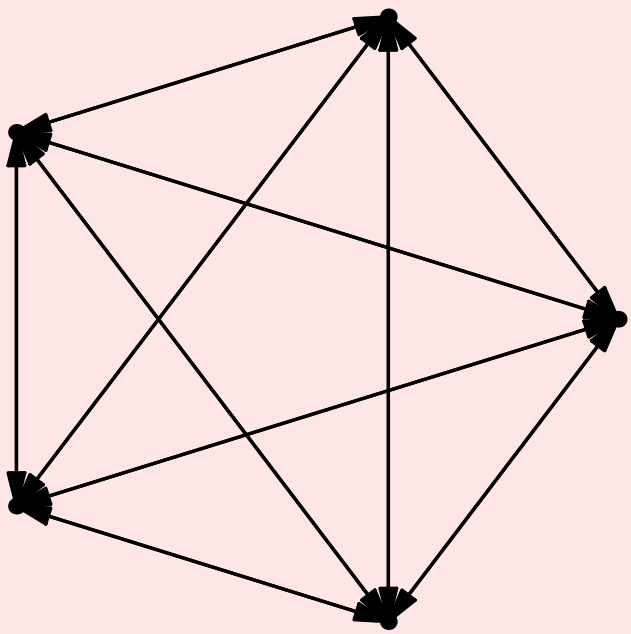
Various functions such as Eccentricity, Diameter, TwoColoring, BipartiteQ, and others now simply call BreadthFirstTraversal with an appropriate argument.

Functions that generate graphs often come with options that allow the user to choose between directed or undirected graphs. Here is an example.

```
?CompleteGraph
```

```
CompleteGraph[n] creates a complete graph on n vertices.An option Type
  that takes on the values Directed or Undirected is allowed.
  The default setting for this option is Type -> Undirected.
  CompleteGraph[a,b,c,...] creates a complete k-partite graph of the
  prescribed shape. The use of CompleteGraph to create a complete
  k-partite graph is obsolete, use CompleteKPartiteGraph instead.
```

```
ShowGraph[ CompleteGraph[5, Type → Directed] ]
```

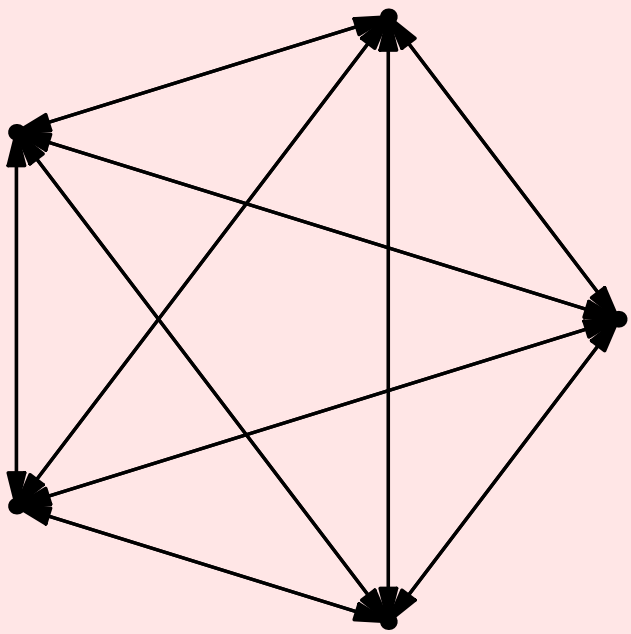


```
- Graphics -
```

```
SetOptions[ CompleteGraph, Type → Directed]
```

```
{Type → Directed}
```

```
ShowGraph[ CompleteGraph[5] ]
```



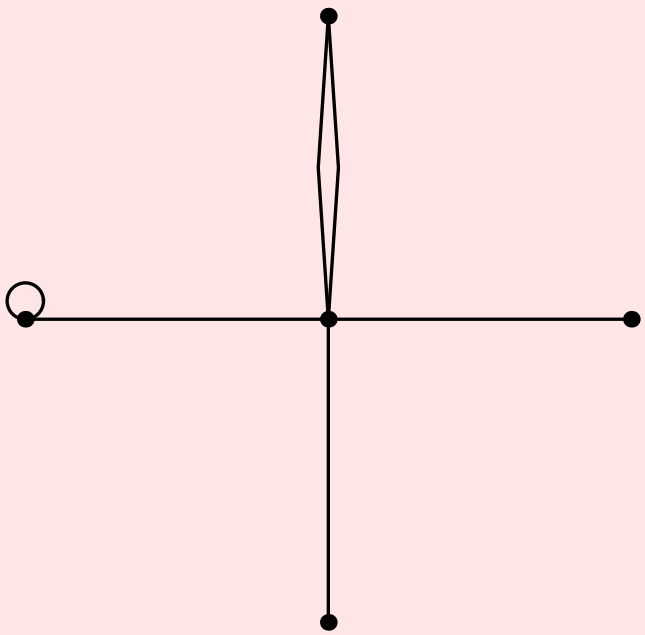
```
- Graphics -
```

Functions that translate between different graph representations now have options that allow the user to pay attention to multiple edges and self–loops.In the following example, the function ToAdjacencyLists, in its default version, makes sure that self–loops and multiple edges show up in the adjacency lists. Using the option Type–>Simple,we can force ToAdjacencyLists to ignore the self–loopsand multiple edges.

```
g = AddEdges[Star[5], {{1, 5}, {2, 2}}]
```

```
-Graph:<6, 5, Undirected>-
```

```
ShowGraph[ g ]
```



```
- Graphics -
```

```
ToAdjacencyLists[g]
```

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

```
ToAdjacencyLists[g, Type → Simple]
```

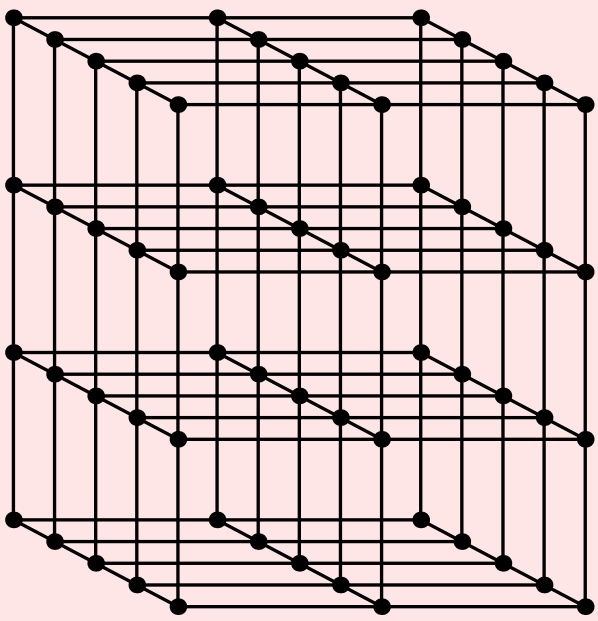
```
{{5}, {5}, {5}, {5}, {1, 2, 3, 4}}
```

New Graph Instances and Classes

```
? GridGraph
```

```
GridGraph[n, m] constructs an n*m grid graph, the product of paths on
  n and m vertices. GridGraph[p, q, r] constructs a p*q*r grid
  graph, the product of GridGraph[p, q] and a path of length r.
```

```
ShowGraph[GridGraph[3, 4, 5] ]
```

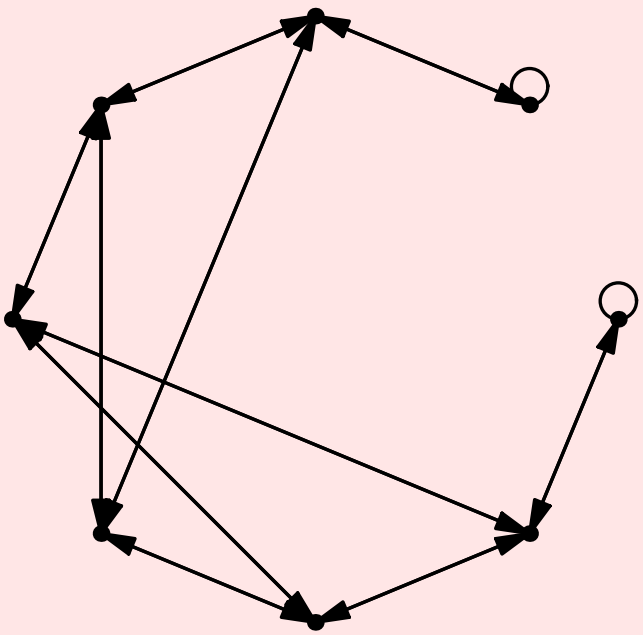


```
- Graphics -
```

```
? ShuffleExchangeGraph
```

```
ShuffleExchangeGraph[n] returns the n-dimensional shuffle-exchange graph
  whose vertices are length n binary strings with an edge from w
  to w' if (i) w' differs from w in its last bit or (ii) w' is
  obtained from w by a cyclic shift left or a cyclic shift right.
```

```
ShowGraph[ ShuffleExchangeGraph[3] ]
```

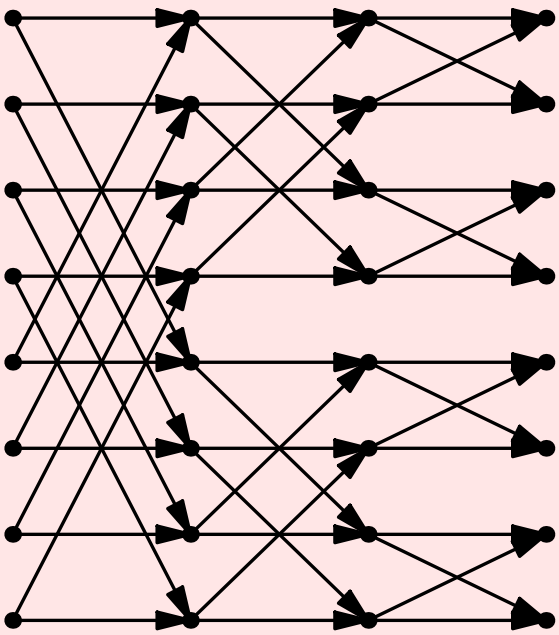


```
- Graphics -
```

```
?ButterflyGraph
```

```
ButterflyGraph[n] returns the n-dimensional Butterfly Graph, a directed
  graph whose vertices are pairs (w, i), where w is a binary string
  of length n and i is an integer in the range 0 through n and whose
  edges go from vertex (w, i) to (w', i+1), if w' is identical to
  w in all bits with the possible exception of the (i+1)th bit.
```

```
ShowGraph[ ButterflyGraph[3] ]
```

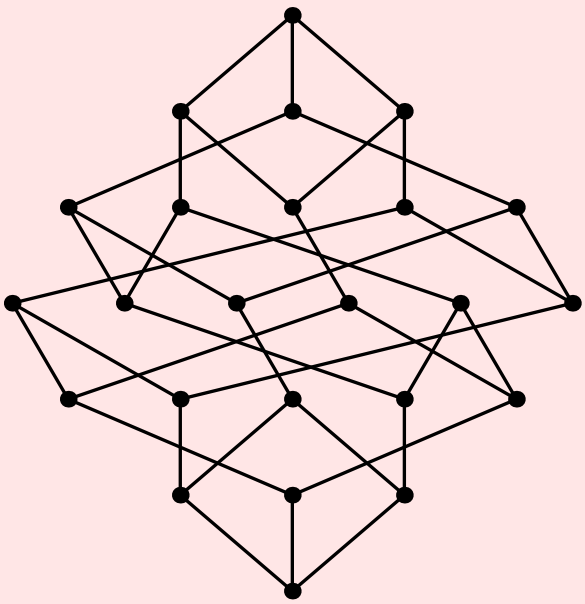


```
- Graphics -
```

```
? InversionPoset
```

```
InversionPoset[n] returns the Hasse diagram of the
  partially ordered set on size-n permutations in which p <
  q if q can be obtained from p by an adjacent transposition
  that places the larger element before the smaller.
```

```
ShowGraph[ InversionPoset[4] ]
```

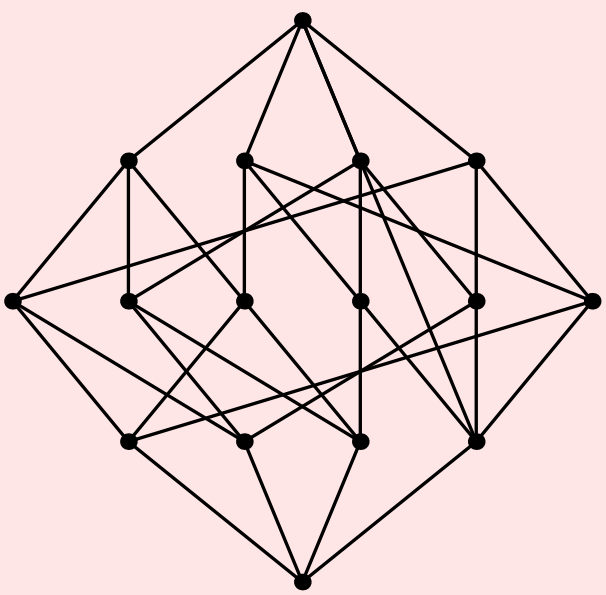


```
- Graphics -
```

```
?BooleanAlgebra
```

```
BooleanAlgebra[n] gives the Hasse
  diagram for the boolean algebra on n elements.
```

```
ShowGraph[ BooleanAlgebra[4] ]
```

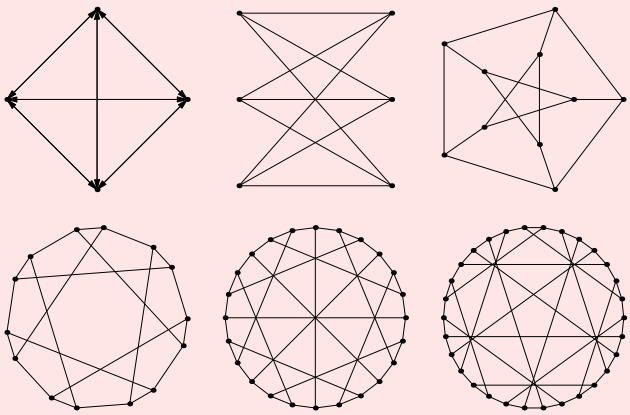


```
- Graphics -
```

```
? CageGraph
```

```
CageGraph[k, r] gives the smallest k-regular graph
  of girth r for certain small values of k and r. CageGraph[
  r] gives CageGraph[3, r]. For k = 3, r can be 3, 4, 5,
  6, 7, 8, or 10. For k = 4 or 5, r can be 3, 4, 5, or 6.
```

```
ShowGraphArray[
 {Table[CageGraph[r], {r, 3, 5}], Table[CageGraph[r], {r, 6, 8}]}]
```



```
- GraphicsArray -
```

Miscellaneous New Functions

Here we show a new function SetEdgeWeights that provides a flexible way of assigning weights to edges in a graph.

```
? SetEdgeWeights
```

```
SetEdgeWeights[g] assigns random real weights in the range [0, 1]
  to edges in g. SetWeights accepts options WeightingFunction
  and WeightRange. WeightingFunction can take values Random,
  RandomInteger, Euclidean, LNorm[n] for non-negative n, or any
  pure function that takes as input two points. WeightRange can
  be an integer range or a real range. The default value for
  WeightingFunction is Random and the default value for WeightRange
  is [0, 1]. SetEdgeWeights[g, e] assigns edge weights to the
  edges in the edge list e. The options WeightingFunction and
  WeightRange apply. SetEdgeWeights[g, w] assigns the weights in the
  weight list w to the edges of g. SetEdgeWeights[g, e, w] assigns
  the weights in the weight list w to the edges in edge list e.
```

```
g = SetEdgeWeights[Wheel[10], WeightingFunction → RandomInteger,
  WeightRange → {3, 5}]; GetEdgeWeights[g]
```

```
{4, 5, 3, 4, 3, 4, 5, 3, 5, 4, 3, 5, 3, 5, 3, 4, 5, 5}
```

```
Options[SetEdgeWeights]
```

```
{WeightingFunction → Random, WeightRange → {0, 1}}
```

```
?SetOptions
```

```
SetOptions[s, name1->value1, name2->value2, ... ] sets the specified
  default options for a symbol s. SetOptions[stream, ... ]
  or SetOptions["name", ... ] sets options associated with
  a particular stream. SetOptions[object, ... ] sets options
  associated with an external object such as a NotebookObject.
```

```
SetOptions[SetEdgeWeights,
 WeightingFunction → RandomInteger, WeightRange → {0, 1}]
```

```
{WeightingFunction → RandomInteger, WeightRange → {0, 1}}
```

```
g = SetEdgeWeights[Wheel[10]]; GetEdgeWeights[g]
```

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

Here is another example of a useful new function. Shortest paths can now be computed even in the presence of negative edge weights.

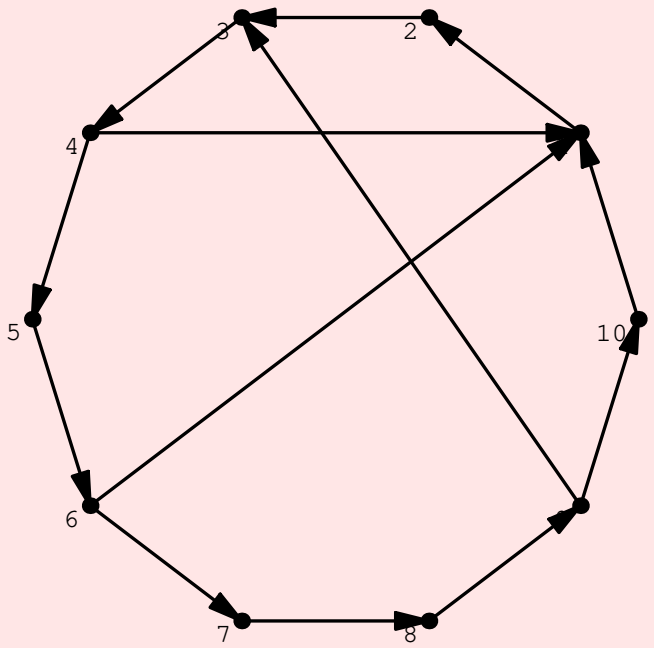
```
?BellmanFord
```

```
BellmanFord[g, v] gives the shortest path spanning tree and associated
  distances from vertex v of graph g. The shortest path spanning tree is
  given by a list in which element i is the predecessor of vertex i in
  the shortest path spanning tree. BellmanFord works correctly even when
  the edge weights are negative, provided there are no negative cycles.
```

```
g = AddEdges[ Cycle[10, Type → Directed], {{4, 1}, {6, 1}, {9, 3}}]
```

```
-Graph:<13, 10, Directed>-
```

```
ShowGraph[g, VertexNumber → On]
```




```
- Graphics -
```

```
g = SetEdgeWeights[g,
  WeightingFunction → RandomInteger, WeightRange → {-4, 5}]
```

```
-Graph:<13, 10, Directed>-
```

```
Edges[g, EdgeWeight]
```

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

```
BellmanFord[g, 1]
```

```
{{4, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {-5., -1., 0., -1., 2., -2., 2., 7., 11., 12.}}
```

```
Dijkstra[g, 1]
```

```
{{4, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {-1, 3, 4, 3, 6, 2, 6, 11, 15, 16}}
```

Another example of a useful new function is TreeIsomorphismQ.

```
?TreeIsomorphismQ
```

```
TreeIsomorphismQ[t1, t2] returns True if
  the trees t1 and t2 are isomorphic; False otherwise.
```

```
g = RandomTree[40]; h = PermuteSubgraph[g, RandomPermutation[40]]
```

```
-Graph:<39, 40, Undirected>-
```

```
TreeIsomorphismQ[g, h]
```

```
True
```

```
IsomorphicQ[g, h]
```

```
True
```

```
{Timing[TreeIsomorphismQ[g, h];], Timing[IsomorphicQ[g, h];]}
```

```
{{0.1 Second, Null}, {1.68 Second, Null}}
```