5. Properties of Graphs

ConnectedComponents

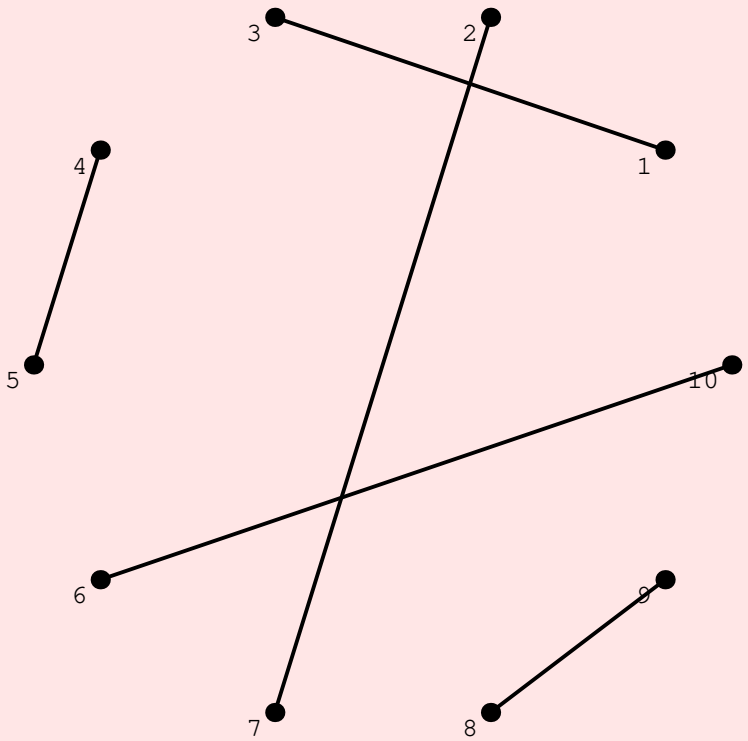
```
? ConnectedComponents
```

```
ConnectedComponents[g] gives the vertices
  of graph g partitioned into connected components.
```

```
ConnectedComponents[s = RegularGraph[1, 10]]
```

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

```
ShowGraph[s, VertexNumber -> On]
```

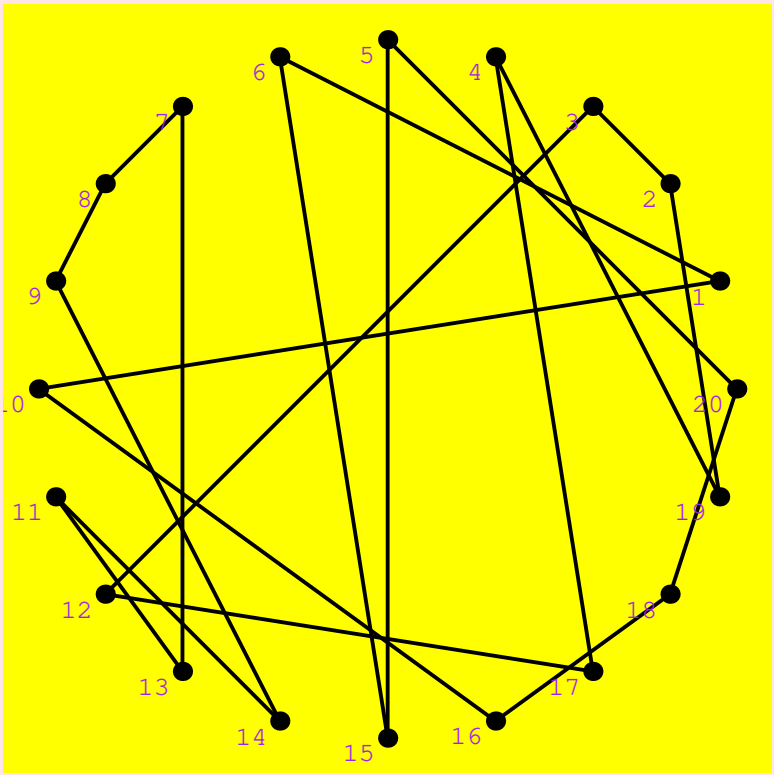



```
- Graphics -
```

```
c = ConnectedComponents[s = RegularGraph[2, 20]]
```

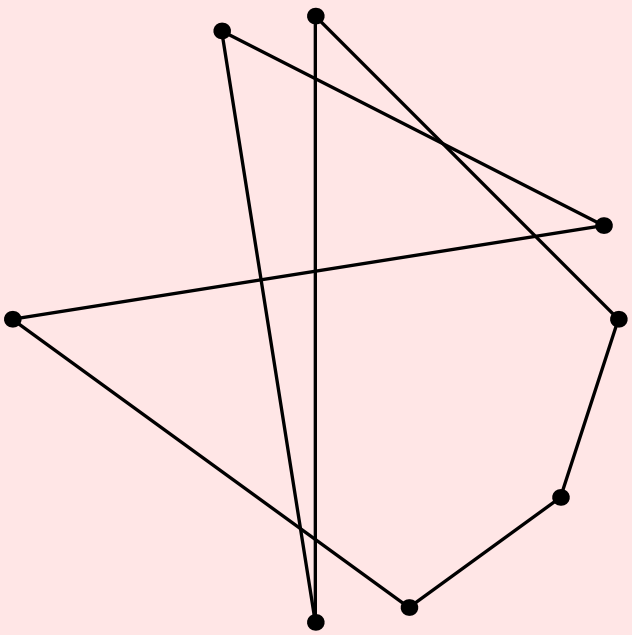
```
{{1, 6, 15, 5, 20, 18, 16, 10}, {2, 3, 12, 17, 4, 19}, {7, 8, 9, 14, 11, 13}}
```

```
ShowGraph[s, VertexNumber -> On,
 VertexNumberColor -> Purple, Background -> Yellow]
```




```
- Graphics -
```

```
ShowGraph[InduceSubgraph[s, c[[1]]]]
```



```
- Graphics -
```

```
a = GraphUnion[Wheel[5], Star[10]];
```

```
ConnectedComponents[ a ]
```

```
{{1, 2, 3, 4, 5}, {6, 15, 7, 8, 9, 10, 11, 12, 13, 14}}
```

```
a = GraphUnion [10, Star[10]]; ConnectedComponents[ a ]
```

```
{{1, 10, 2, 3, 4, 5, 6, 7, 8, 9}, {11, 20, 12, 13, 14, 15, 16, 17, 18, 19},
 {21, 30, 22, 23, 24, 25, 26, 27, 28, 29}, {31, 40, 32, 33, 34, 35, 36, 37, 38, 39},
 {41, 50, 42, 43, 44, 45, 46, 47, 48, 49}, {51, 60, 52, 53, 54, 55, 56, 57, 58, 59},
 {61, 70, 62, 63, 64, 65, 66, 67, 68, 69}, {71, 80, 72, 73, 74, 75, 76, 77, 78, 79},
 {81, 90, 82, 83, 84, 85, 86, 87, 88, 89}, {91, 100, 92, 93, 94, 95, 96, 97, 98, 99}}
```

TIMING DISCUSSION

First we have a comparison between the new and old implementations of ConnectedComponents. As can be seen from the tables of timings given below, the new implementation is dramatically faster than the old implementation –factor of speed ranges from 80 to more than 100.

```
$RecursionLimit = 100000;
```

```
g = Table[DiscreteMath`OldCombinatorica`RandomGraph[i * 100, 0.01], {i, 5}];
```

```
h = Table[RandomGraph[i * 100, 0.01], {i, 10}]
```

```
{-Graph:<52, 100, Undirected>-, -Graph:<194, 200, Undirected>-,
 -Graph:<433, 300, Undirected>-, -Graph:<812, 400, Undirected>-,
 -Graph:<1264, 500, Undirected>-, -Graph:<1804, 600, Undirected>-,
 -Graph:<2386, 700, Undirected>-, -Graph:<3313, 800, Undirected>-,
 -Graph:<4034, 900, Undirected>-, -Graph:<4922, 1000, Undirected>-}
```

```
Table[{Length[DiscreteMath`OldCombinatorica`ConnectedComponents[g[[i]]]],
  Length[ConnectedComponents[h[[i]]]]}, {i, 5}]
```

```
{{51, 49}, {42, 26}, {15, 13}, {12, 8}, {3, 3}}
```

```
ort = Table[
  Timing[DiscreteMath`OldCombinatorica`ConnectedComponents[g[[i]]];], {i, 5}
```

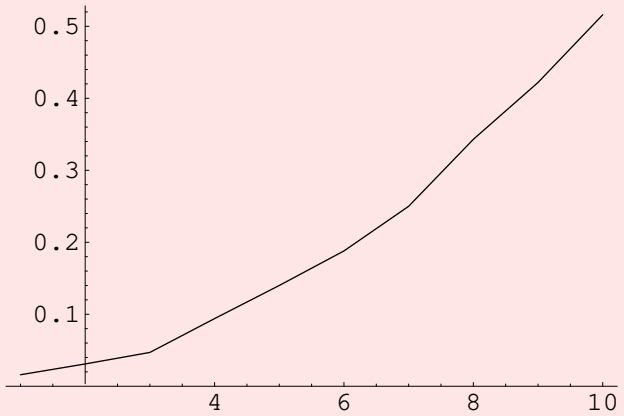
```
{{2.407 Second, Null}, {7.593 Second, Null},
 {6.282 Second, Null}, {8.75 Second, Null}, {3.562 Second, Null}}
```

```
rt = Table[ Timing[ ConnectedComponents[ h[[i]] ]; ], {i, 10}]
```

```
{{0.016 Second, Null}, {0.031 Second, Null},
 {0.047 Second, Null}, {0.094 Second, Null},
 {0.14 Second, Null}, {0.188 Second, Null}, {0.25 Second, Null},
 {0.343 Second, Null}, {0.422 Second, Null}, {0.516 Second, Null}}
```

The running time plot for ConnectedComponents confirms that the running time grows quadratically in the size of the graph. However, the experiment might be misleading since the expected number of edges in the random graph also grows quadratically. So I do a second experiment below. In that experiment the running time seems almost linear! Note that in this experiment, the number of vertices and the number of edges are growing in a roughly linear fashion.

```
p1 = ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
gt = Table[ InduceSubgraph[GridGraph[20, 10 * i],
    RandomKSubset[ Range[ 200 * i], 100 * i]], {i, 10}];
```

```
Map[ V, gt]
```

```
{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000}
```

```
Map[M, gt]
```

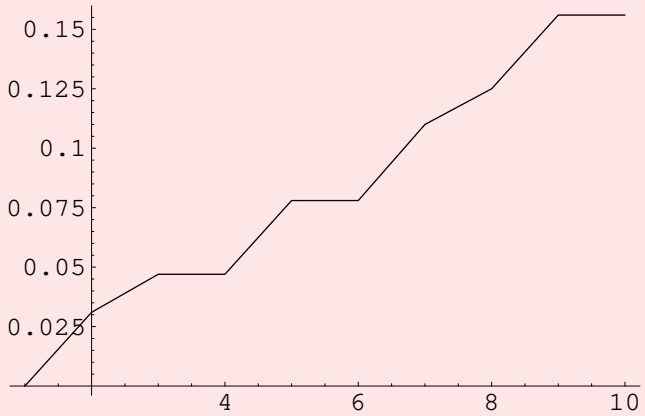
```
{92, 210, 287, 369, 475, 588, 700, 791, 895, 974}
```

```
rt = Table[ Timing[ ConnectedComponents[ gt[[i]] ];], {i, 10}]
```

```
{{0. Second, Null}, {0.031 Second, Null},
 {0.047 Second, Null}, {0.047 Second, Null},
 {0.078 Second, Null}, {0.078 Second, Null}, {0.11 Second, Null},
 {0.125 Second, Null}, {0.156 Second, Null}, {0.156 Second, Null}}
```

```
Clear[gt]
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

The following experiment shows that BFS may be faster than DFS and it might be better to use BFS in the new ConnectedComponents. However, I will wait to make this change to ConnectedComponents because I expect both DFS and BFS to be speeded up.

```
s = GridGraph[50, 50];
{Timing[DepthFirstTraversal[s, 1];], Timing[BreadthFirstTraversal[s, 1 ];]}
```

```
{{2.547 Second, Null}, {1.235 Second, Null}}
```

Here is a larger experiment with ConnectedComponents.

```
s = InduceSubgraph[GridGraph[70, 70], RandomSubset[Range[4900]]]
```

```
-Graph:<2424, 2459, Undirected>-
```

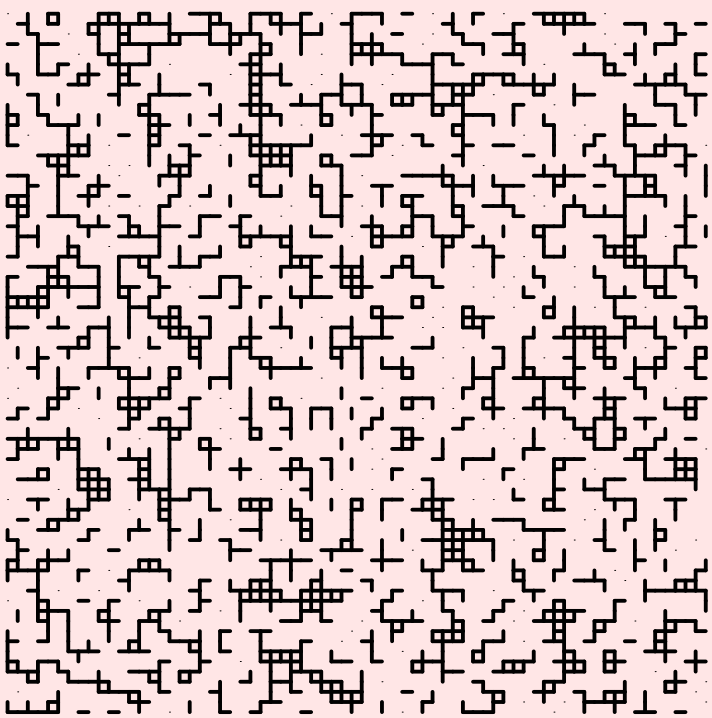
```
V[s]
```

```
2459
```

```
M[s]
```

```
2424
```

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



```
- Graphics -
```

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

```
{0.562 Second, Null}
```

```
Length[c]
```

```
343
```

```
Max[Map[Length, c]]
```

```
143
```

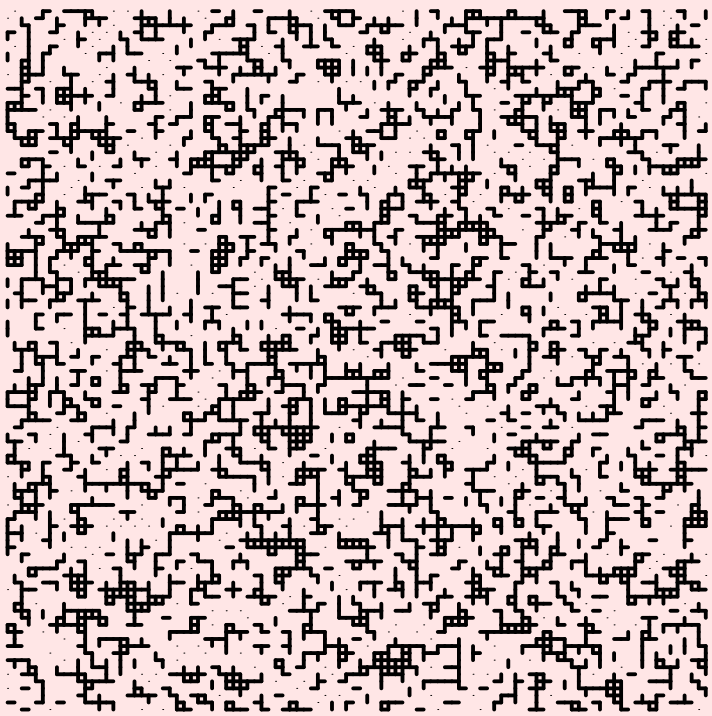
Only 11 seconds for a graph with 5030 vertices and lots of components!

```
g = GridGraph[100, 100]
```

```
-Graph:<19800, 10000, Undirected>-
```

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

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



```
- Graphics -
```

```
V[h]
```

```
4940
```

```
M[h]
```

```
4827
```

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

```
{1.531 Second, Null}
```

```
{Length[c], Max[Map[Length, c]]}
```

```
{729, 177}
```

ConnectedQ

```
?ConnectedQ
```

```
ConnectedQ[g] yields True if undirected graph g is connected. If g is
  directed, the function returns True if the underlying undirected graph
  is connected. ConnectedQ[g, Strong] and ConnectedQ[g, Weak] yield True
  if the directed graph g is strongly or weakly connected, respectively.
```

```
ConnectedQ[Wheel[10]]
```

```
True
```

```
ConnectedQ[DeleteVertices[Star[10], {10}]]
```

```
False
```

```
ConnectedQ[Wheel[10], Weak]
```

```
ConnectedQ[-Graph:<18, 10, Undirected>-, Weak]
```

```
ConnectedQ[OrientGraph[Wheel[10]], Strong]
```

```
True
```

```
ConnectedQ[SetGraphOptions[RandomTree[20], EdgeDirection → On], Strong]
```

```
False
```

```
ConnectedQ[SetGraphOptions[RandomTree[20], EdgeDirection → On], Weak]
```

```
True
```

NOTES

* ConnectedQ with the strong and weak options works only on directed graphs.

```
ConnectedQ[ Wheel[10], Weak]
```

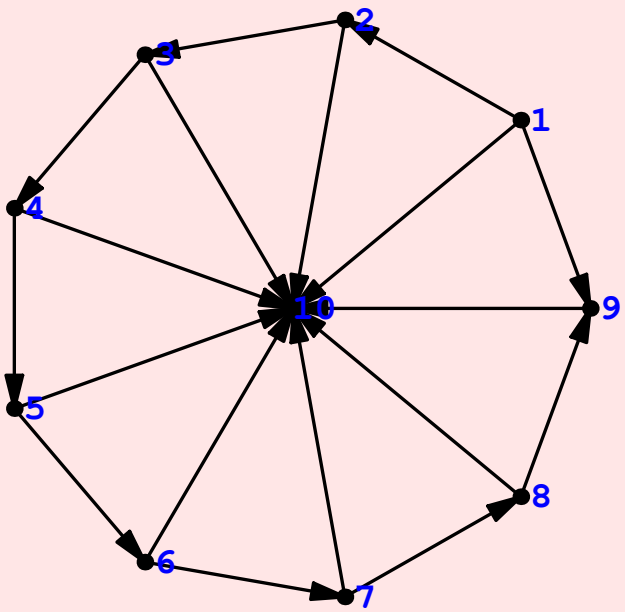
```
ConnectedQ[-Graph:<18, 10, Undirected>-, Weak]
```

```
ConnectedQ[ Wheel[10], Strong]
```

```
ConnectedQ[-Graph:<18, 10, Undirected>-, Strong]
```

```
g = SetGraphOptions[ Wheel[10], EdgeDirection -> On];
```

```
ShowGraph[ g , VertexNumber -> Text[ {0.03, 0},
   TextStyle -> {FontSize -> 14, FontColor -> Blue, FontWeight -> Bold}],
 PlotRange -> Large[0.05]]
```




```
- Graphics -
```

```
ConnectedQ[ g, Weak]
```

```
True
```

```
ConnectedQ[g, Strong]
```

```
False
```

```
h = AddEdges[g, {{{9, 1}}, {{10, 1}}}];
```

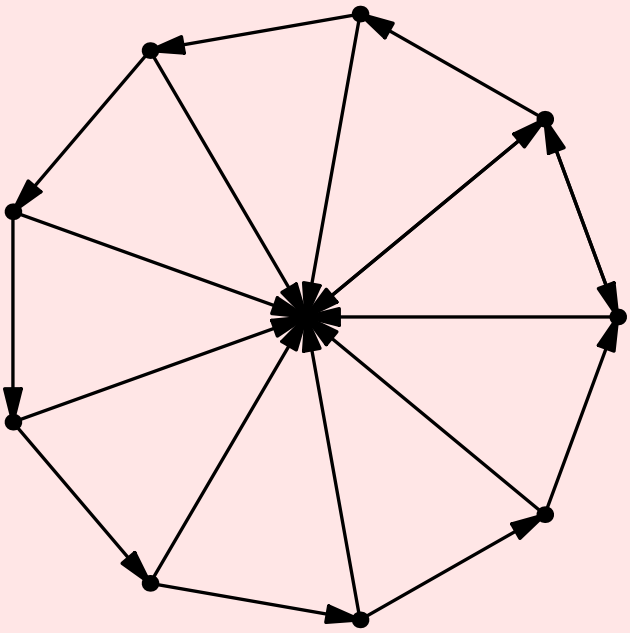
```
ConnectedQ[h, Strong]
```

```
True
```

```
ConnectedQ[h, Weak]
```

```
True
```

```
ShowGraph[ h ]
```



```
- Graphics -
```

WeaklyConnectedComponents, StronglyConnectedComponents

```
?WeaklyConnectedComponents
```

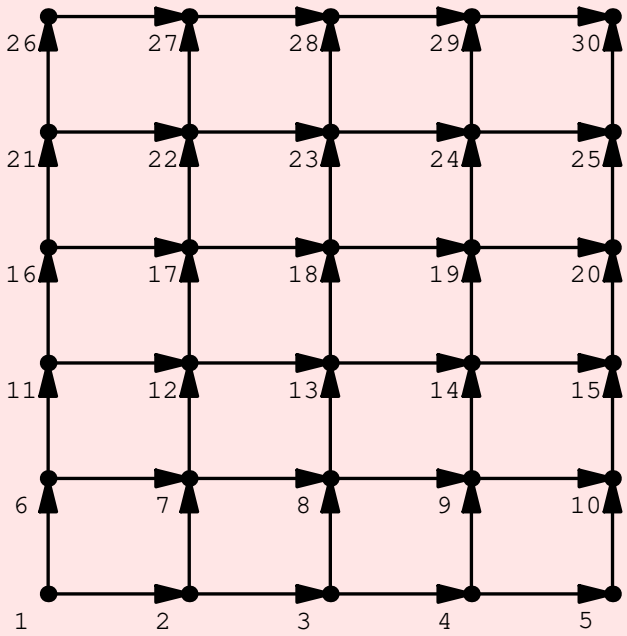
```
WeaklyConnectedComponents[g] gives the weakly connected
  components of directed graph g as lists of vertices.
```

```
?StronglyConnectedComponents
```

```
StronglyConnectedComponents[g] gives the strongly
  connected components of directed graph g as lists of vertices.
```

```
s = SetGraphOptions[GridGraph[5, 6], EdgeDirection -> On];
```

```
ShowGraph[s, VertexNumber -> Text[{-0.04, -0.04}], PlotRange -> Large[0.05]]
```




```
- Graphics -
```

```
WeaklyConnectedComponents[s]
```

```
{{1, 2, 3, 4, 5, 10, 9, 8, 7, 6, 11, 12, 13, 14, 15,
  20, 19, 18, 17, 16, 21, 22, 23, 24, 25, 30, 29, 28, 27, 26}}
```

```
StronglyConnectedComponents[s]
```

```
{{30}, {25}, {20}, {15}, {10}, {5}, {29}, {24},
 {19}, {14}, {9}, {4}, {28}, {23}, {18}, {13}, {8}, {3}, {27},
 {22}, {17}, {12}, {7}, {2}, {26}, {21}, {16}, {11}, {6}, {1}}
```

```
s = AddEdges[s, {{{30, 1}}}];
```

```
StronglyConnectedComponents[ s ]
```

```
{{26, 21, 16, 11, 6, 27, 22, 17, 12, 7, 28, 23, 18,
  13, 8, 29, 24, 19, 14, 9, 30, 25, 20, 15, 10, 5, 4, 3, 2, 1}}
```

NOTES

* The above example shows that s is almost strongly connected −addition of one edge makes it strongly connected.

TIMING DISCUSSION

WeaklyConnectedComponents simply turns the graph into an undirected graph and runs the function ConnectedComponents on that. Therefore its running time is completely dominated by the running time of ConnectedComponents.

Here I focus on the running time of StronglyConnectedComponents. As the plot below shows StronglyConnectedComponents has running time that is roughly quadratic in the size of the graph. This should not be the case and I should try to recode StronglyConnectedComponents so that it runs in linear time.

TO DO

* Redo strongly connected components so that it runs in linear time.

The timings and plots further below make it clear that the new implementation of StronglyConnectedComponents[...] is much faster than the old implementation. The plots seem to indicate that both functions take quadratic time with different constants.

```
gt = Table[ RandomGraph[100 i, N[1 / 100 i], Directed], {i, 10}];
```

```
rt = Table[ Timing[StronglyConnectedComponents[gt[[i]]];], {i, 10}]
```

```
{{0.031 Second, Null}, {0.078 Second, Null},
 {0.203 Second, Null}, {0.453 Second, Null},
 {0.906 Second, Null}, {1.641 Second, Null}, {2.734 Second, Null},
 {4.329 Second, Null}, {6.609 Second, Null}, {9.375 Second, Null}}
```

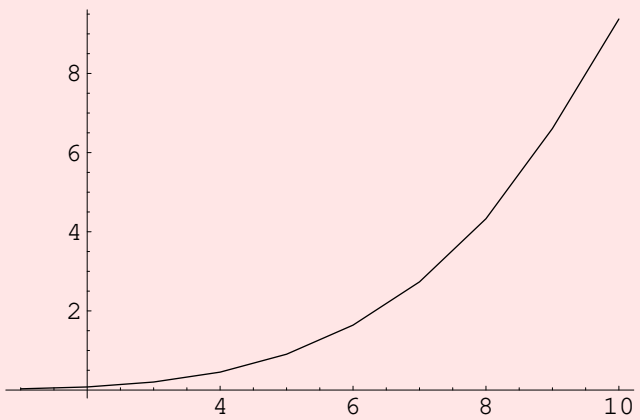
```
Map[V, gt]
```

```
{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000}
```

```
Map[M, gt]
```

```
{105, 774, 2681, 6422, 12569, 21526, 33894, 51405, 72820, 99907}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
ogt = Table[DiscreteMath`OldCombinatorica`RandomGraph[
    100 i, N[1 / 100 i], {0, 1}], {i, 5}];
```

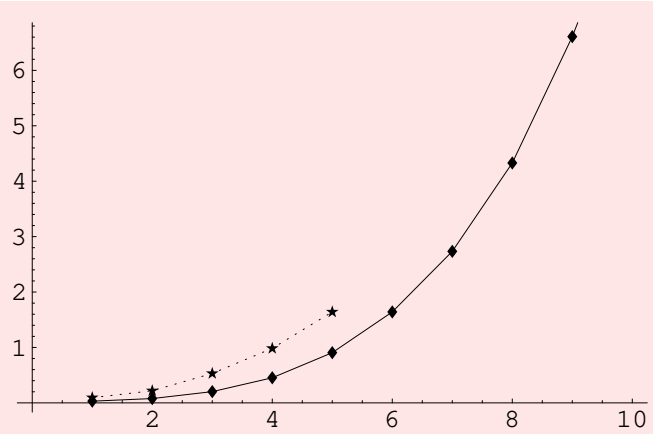
```
ort =
 Table[ Timing[ DiscreteMath`OldCombinatorica`StronglyConnectedComponents[
     ogt[[i]] ];], {i, 5}]
```

```
{{0.094 Second, Null}, {0.219 Second, Null},
 {0.531 Second, Null}, {0.984 Second, Null}, {1.641 Second, Null}}
```

```
Clear[MultipleListPlot]; Remove[ MultipleListPlot];
```

```
<< Graphics`MultipleListPlot`
```

```
MultipleListPlot[Map[#[[1, 1]] &, rt],
 Map[ #[[1, 1]] &, ort], PlotJoined -> True]
```




```
- Graphics -
```

```
g = SetGraphOptions[GridGraph[70, 70], EdgeDirection -> On];
```

```
{Timing[cw = WeaklyConnectedComponents[g];],
 Timing[cs = StronglyConnectedComponents[g];], Length[cw], Length[cs]}
```

```
{{12.156 Second, Null}, {5.5 Second, Null}, 1, 4900}
```

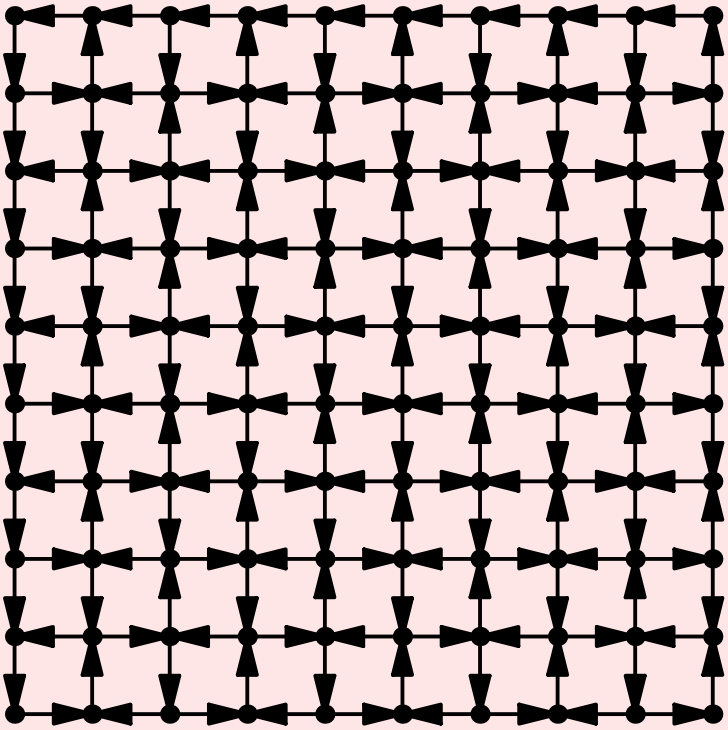
OrientGraph

```
?OrientGraph
```

```
OrientGraph[g] assigns a direction to each edge of a bridgeless,
  undirected graph g, so that the graph is strongly connected.
```

```
g = GridGraph[10, 10];
```

```
ShowGraph[ h = OrientGraph[ GridGraph[10, 10] ] ]
```



```
- Graphics -
```

```
Length[StronglyConnectedComponents[h]]
```

```
1
```

```
g = RandomGraph[20, .6];
```

```
Bridges[g]
```

```
{}
```

```
Length[ StronglyConnectedComponents[SetGraphOptions[g, EdgeDirection -> On]] ]
```

```
20
```

```
h = OrientGraph[g];
```

```
Length[ StronglyConnectedComponents[g] ]
```

```
1
```

TIMING DISCUSSION

OrientGraph in new Combinatorica is extremely slow!! About 227 seconds for 400 vertex grid graph! Most of the time is however taken by ExtractCycles. So ExtractCycles is in need of major repair. One immediate improvement is for ExtractCycles to not compute the new adjacency list representation from scratch after each cycle has been deleted. Of course, OrientGraph in "old" Combinatorica seems 4 times slower (based on one sample point).

I wonder if it is possible to do this in linear time. Even if it isn't, I am sure it is possible to do this in much more speedy fashion.

TO DO

Speed up ExtractCycles[...]

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

```
Timing [ OrientGraph[g];]
```

```
{49.906 Second, Null}
```

```
Timing[ ExtractCycles[g]; ]
```

```
{48.813 Second, Null}
```

```
h = DiscreteMath`OldCombinatorica`GridGraph[20, 20];
Timing[ DiscreteMath`OldCombinatorica`OrientGraph[h]; ]
```

```
{181.047 Second, Null}
```

BiconnectedComponents

```
?BiconnectedComponents
```

```
BiconnectedComponents[g] gives a list of the biconnected components of
  graph g. If g is directed, the underlying undirected graph is used.
```

```
BiconnectedComponents[RandomTree[100]]
```

```
{{1, 5}, {28, 84}, {6, 51}, {2, 19}, {14, 19}, {14, 51}, {85, 99}, {48, 85},
 {48, 75}, {13, 76}, {15, 76}, {15, 63}, {63, 75}, {45, 75}, {8, 45}, {8, 24},
 {23, 24}, {23, 51}, {51, 89}, {18, 59}, {30, 43}, {38, 43}, {86, 90}, {33, 86},
 {27, 33}, {27, 41}, {27, 57}, {38, 57}, {38, 67}, {38, 78}, {12, 87},
 {12, 98}, {7, 98}, {7, 74}, {35, 65}, {16, 65}, {16, 74}, {21, 39}, {26, 53},
 {49, 64}, {26, 64}, {26, 83}, {55, 60}, {3, 50}, {31, 40}, {40, 50}, {50, 70},
 {73, 100}, {50, 100}, {50, 80}, {20, 80}, {20, 56}, {56, 93}, {61, 93},
 {55, 93}, {55, 83}, {39, 83}, {39, 74}, {22, 74}, {22, 77}, {11, 91}, {34, 69},
 {44, 47}, {47, 69}, {52, 66}, {32, 58}, {36, 82}, {58, 82}, {54, 58},
 {54, 66}, {66, 69}, {88, 96}, {69, 88}, {17, 69}, {17, 72}, {72, 91},
 {77, 91}, {77, 78}, {59, 78}, {59, 89}, {9, 46}, {9, 68}, {9, 25}, {37, 81},
 {37, 71}, {25, 71}, {25, 42}, {42, 79}, {10, 95}, {10, 97}, {10, 92},
 {79, 92}, {29, 79}, {29, 62}, {62, 89}, {84, 89}, {1, 84}, {4, 94}, {1, 94}}
```

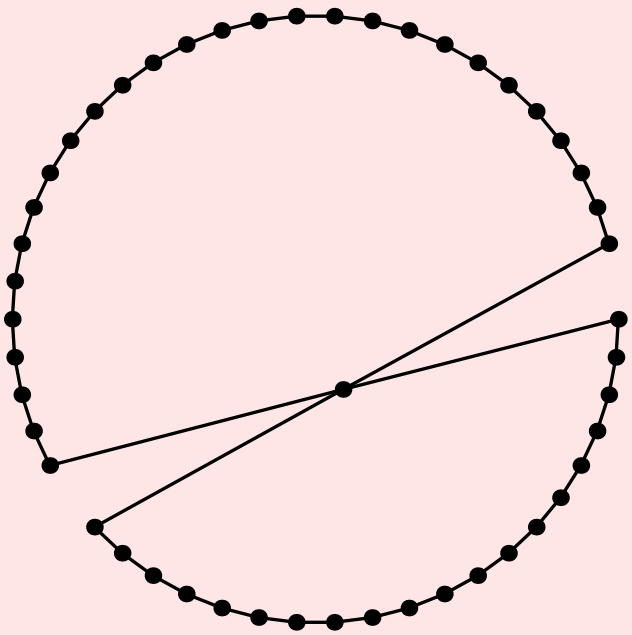
```
Length[%]
```

```
99
```

```
BiconnectedComponents[Cycle[100]]
```

```
{{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
  24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
  44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
  63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
  82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}}
```

```
ShowGraph[g = Contract[Cycle[50], {1, 30}]]
```



```
- Graphics -
```

```
BiconnectedComponents[g]
```

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

TIMING DISCUSSION

Like ConnectedComponents, StronglyConnectedComponents etc, BiconnectedComponents is also quadratic in running time. This is not clear from the plot below. I wonder what is giong on.

Comparing the running times of the old and new implementation reveals that the new implementation is dramatically faster! For a 200−vertexrandom graph with p =1/200 the new implementation takes 0.21 s while the old implementation takes 88.62 s! When the two timing tables are plotted in the same plot, the plot for the new implementation is not visible.

```
gt = Table[ RandomGraph[i * 10, N[1 / (10 i)]], {i, 20}];
```

```
rt = Table[Timing[ BiconnectedComponents[ gt[[i]] ];], {i, 20}]
```

```
{{0. Second, Null}, {0.015 Second, Null},
 {0. Second, Null}, {0.016 Second, Null}, {0.016 Second, Null},
 {0.015 Second, Null}, {0.016 Second, Null}, {0.031 Second, Null},
 {0.031 Second, Null}, {0.032 Second, Null}, {0.031 Second, Null},
 {0.031 Second, Null}, {0.047 Second, Null}, {0.047 Second, Null},
 {0.047 Second, Null}, {0.047 Second, Null}, {0.062 Second, Null},
 {0.047 Second, Null}, {0.062 Second, Null}, {0.063 Second, Null}}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```

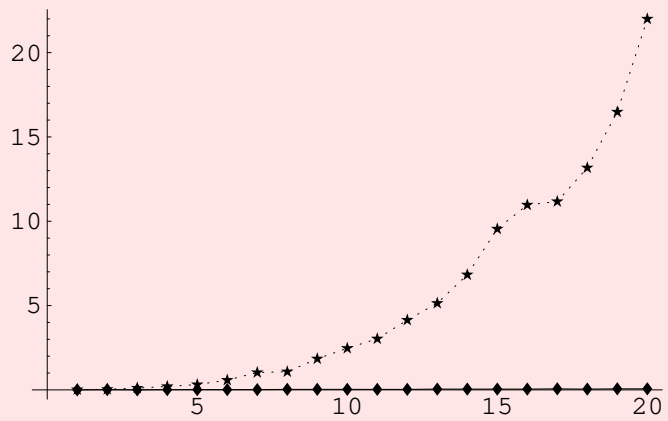
```
- Graphics -
```

```
gt =
  Table[ DiscreteMath`OldCombinatorica`RandomGraph[i * 10, 1 / (10 i)], {i, 20}];
```

```
ort = Table[Timing[
   DiscreteMath`OldCombinatorica`BiconnectedComponents[ gt[[i]] ];], {i, 20}]
```

```
{{0.016 Second, Null}, {0.031 Second, Null},
 {0.109 Second, Null}, {0.203 Second, Null}, {0.313 Second, Null},
 {0.578 Second, Null}, {1.031 Second, Null}, {1.078 Second, Null},
 {1.844 Second, Null}, {2.469 Second, Null}, {3.031 Second, Null},
 {4.141 Second, Null}, {5.14 Second, Null}, {6.829 Second, Null},
 {9.546 Second, Null}, {10.969 Second, Null}, {11.172 Second, Null},
 {13.172 Second, Null}, {16.484 Second, Null}, {22. Second, Null}}
```

```
MultipleListPlot[ Map[ #[[1, 1]] &, rt],
 Map[ #[[1, 1]] &, ort], PlotJoined -> True ]
```




```
- Graphics -
```

TO DO

* Also we should definitely produce the Block–Cutpoint–Tree of a graph. It would be a nice illustration of the excellent properties that blocks and cutpoints have.

* Here we might also consider discussing the connectivity approximation algorithms of Khuller. These are easily implemented and so it might not be a bad idea to include a few of these.

BiconnectedQ

```
?BiconnectedQ
```

```
BiconnectedQ[g] yields True if graph g is biconnected.
  If g is directed, the underlying undirected graph is used.
```

```
g = GridGraph[4, 4];
```

```
BiconnectedQ[g]
```

```
True
```

```
BiconnectedQ[CompleteGraph[10]]
```

```
True
```

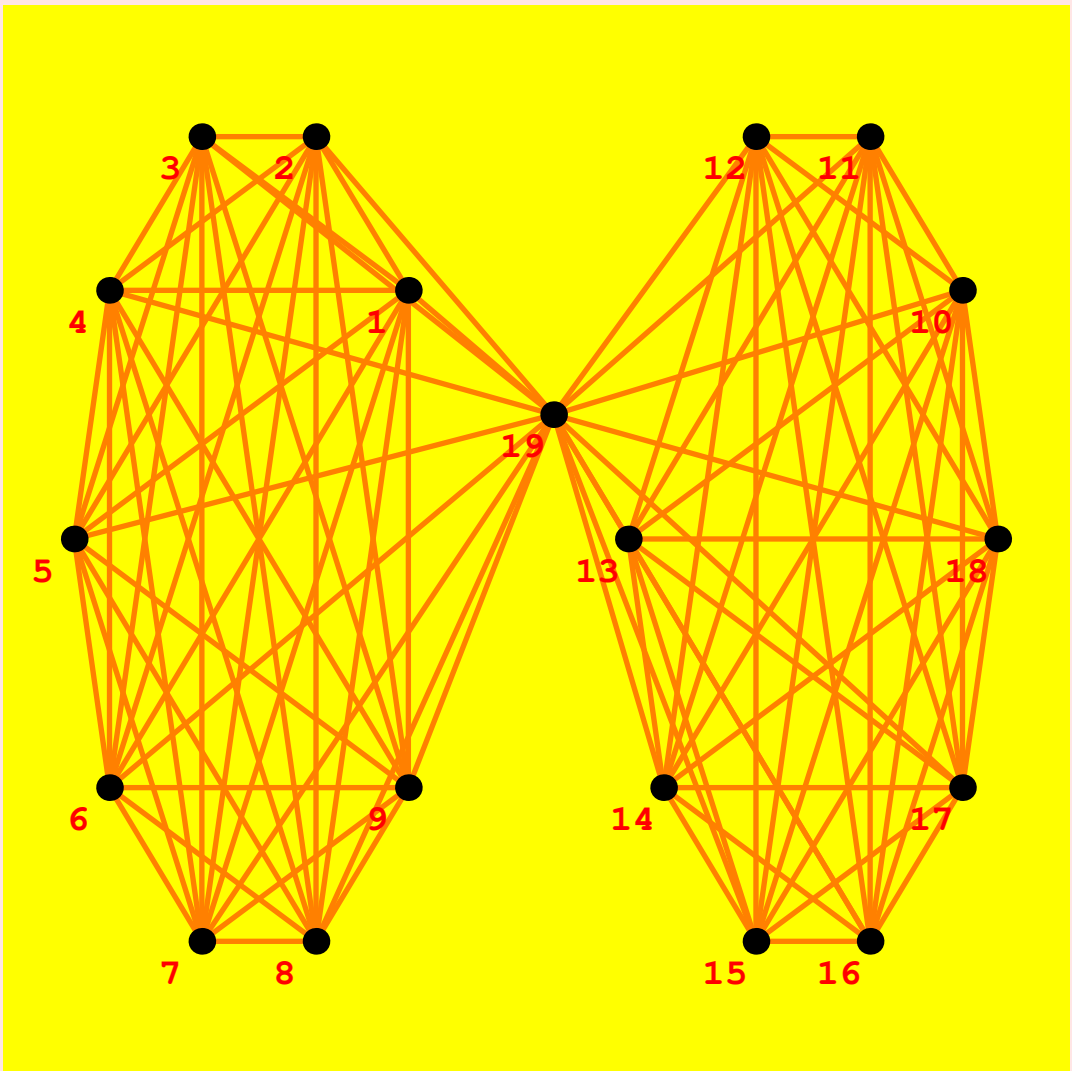
```
BiconnectedQ[GraphUnion[CompleteGraph[10], CompleteGraph[10]]]
```

```
False
```

```
BiconnectedQ[
 g = Contract[GraphUnion[CompleteGraph[10], CompleteGraph[10]], {10, 14}]]
```

```
False
```

```
ShowGraph[g, VertexNumber -> Text[{-0.03, -0.03},
   TextStyle -> {FontSize -> 14, FontWeight -> Bold, FontColor -> Red}],
 Background -> Yellow, EdgeColor -> Orange, ImageSize -> 400,
 PlotRange -> Large[0.05]]
```




```
- Graphics -
```

NOTES

* This function simply finds all the biconnected components by calling the new BiconnectedComponents and then checks the number of biconnected components found. So its running time will be almost identical to the running time of BiconnectedComponents. The same is true for the next two functions: ArticulationVertices and Bridges.

ArticulationVertices

```
?ArticulationVertices
```

```
ArticulationVertices[g] gives a list of all articulation vertices in
  graph g. These are vertices whose removal will disconnect the graph.
```

```
g = Star[20];
```

```
ArticulationVertices[g]
```

```
{20}
```

```
ArticulationVertices[CompleteGraph[10]]
```

```
{}
```

Bridges

```
?Bridges
```

```
Bridges[g] gives a list of the bridges of graph g,
  that is, the edges whose removal disconnects the graph.
```

```
Bridges[RandomTree[10]]
```

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

```
g = GridGraph[10, 10];
```

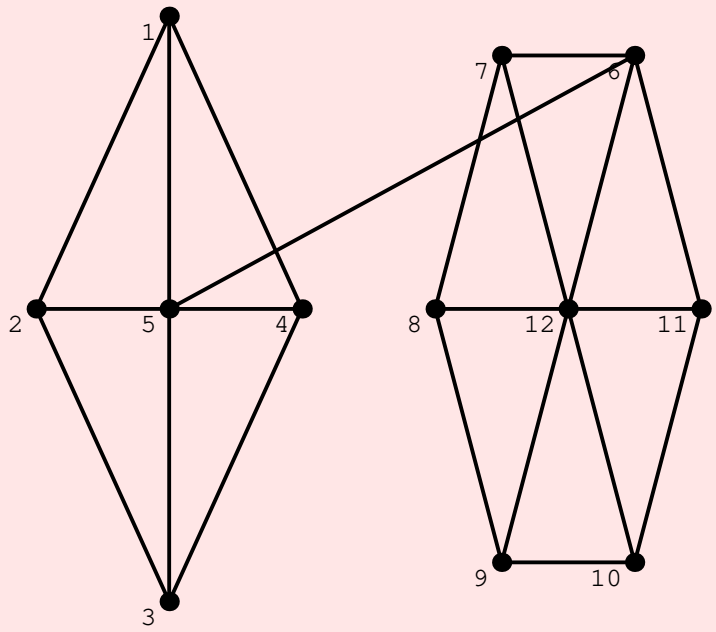
```
Bridges[g]
```

```
{}
```

```
Bridges[s = AddEdges[GraphUnion[Wheel[5], Wheel[7]], {{{5, 6}}}]]
```

```
{{5, 6}}
```

```
ShowGraph[s, VertexNumber -> On, PlotRange -> Large[0.05]]
```




```
- Graphics -
```

EdgeConnectivity

```
?EdgeConnectivity
```

```
EdgeConnectivity[g] gives the minimum number
  of edges whose deletion from graph g disconnects it.
```

```
EdgeConnectivity[ GridGraph[20, 5]]
```

```
2
```

```
EdgeConnectivity[s = AddEdges[GraphUnion[Wheel[5], Wheel[7]], {{{5, 6}}}]]
```

```
1
```

```
EdgeConnectivity[ RandomGraph[20, .3] ]
```

```
1
```

```
EdgeConnectivity[ RandomGraph[20, .5] ]
```

```
6
```

```
EdgeConnectivity[ RandomGraph[20, .7] ]
```

```
10
```

```
EdgeConnectivity [ CompleteGraph[10] ]
```

```
9
```

TIMING DISCUSSION

EdgeConnectivity is extremely slow because it calls NetworkFlow n times. 220 seconds for a 5 X 250 grid graph!! This needs substantial repair. The way to do EdgeConnectivity efficiently would be to implement the Stoer−Wagneralgorithm. I should also implement preflow−pushto speedup the NetworkFlow algorithm. I should then do a timing comparison of both algorithms.

The new implementation is faster than the old implementation, but this is no reason to gloat given that both implementations are terribly slow.

TO DO

* Implement Stoer−Wagneralgorithm for edge connectivity.

* Implement the pre−flowpuch algorithm for Network flows.

```
gt = Table [RandomGraph[5 i, N[1 / (5 i)]], {i, 15}];
```

```
rt = Table [Timing[ EdgeConnectivity[ gt[[i]] ]; ], {i, 15}]
```

```
{{0. Second, Null}, {0.047 Second, Null}, {0.047 Second, Null},
 {0.109 Second, Null}, {0.125 Second, Null}, {0.188 Second, Null},
 {0.25 Second, Null}, {0.469 Second, Null}, {0.453 Second, Null},
 {0.875 Second, Null}, {0.578 Second, Null}, {0.812 Second, Null},
 {0.938 Second, Null}, {1.031 Second, Null}, {1.063 Second, Null}}
```

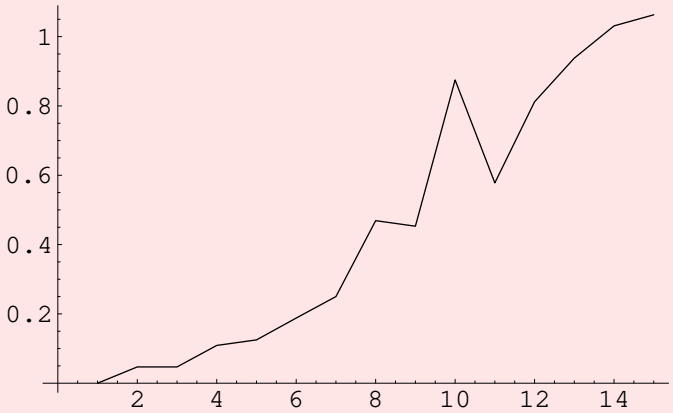
```
Map[V, gt]
```

```
{5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75}
```

```
Map[M, gt]
```

```
{1, 6, 5, 10, 11, 13, 14, 26, 25, 36, 26, 40, 43, 44, 37}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```



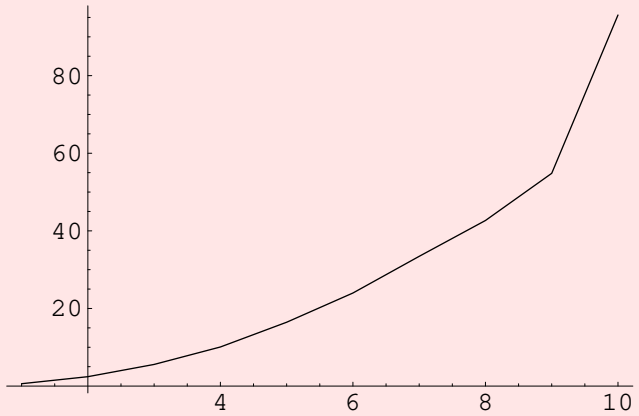

```
- Graphics -
```

```
gt = Table [GridGraph[5, 5 i], {i, 10}];
```

```
rt = Table [Timing[ EdgeConnectivity[ gt[[i]] ]; ], {i, 10}]
```

```
{{0.609 Second, Null}, {2.422 Second, Null},
 {5.578 Second, Null}, {10.078 Second, Null}, {16.453 Second, Null},
 {23.985 Second, Null}, {33.437 Second, Null},
 {42.672 Second, Null}, {54.828 Second, Null}, {95.625 Second, Null}}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
h = Table [DiscreteMath`OldCombinatorica`RandomGraph[i * 5, N[1 / 5 i]], {i, 5}]
```

```
Table [
 Timing[ DiscreteMath`OldCombinatorica`EdgeConnectivity[ h[[i]] ]; ], {i, 5}]
```

```
{{0.015 Second, Null}, {0.094 Second, Null},
 {0.625 Second, Null}, {2. Second, Null}, {4.375 Second, Null}}
```

VertexConnectivity

```
? VertexConnectivity
```

```
VertexConnectivity[g] gives the minimum number
  of vertices whose deletion from graph g disconnects it.
```

```
VertexConnectivity[ GridGraph[10, 3] ]
```

```
2
```

```
VertexConnectivity[ Star[30] ]
```

```
1
```

```
VertexConnectivity[ CompleteGraph[4] ]
```

```
3
```

```
DiscreteMath`OldCombinatorica`VertexConnectivity [
 DiscreteMath`OldCombinatorica`CompleteGraph [ 4] ]
```

```
4
```

NOTES

* The convention is to define the vertex−connectivityof a complete graph of n vertices as n−1.Old Combinatorica does not get this right.

TIMING DISCUSSION

Vertex connectivity is too slow −that fact that the old version of this function is even slower is not a source of comfort. Calling Network Flow n times is a bad idea and an overkill. As I write in the comments for EdgeConnectivity, I should implement the Stoer−Wagneralgorithm for both edge and vertex connectivity. Of course, I should also speed up network flow.

```
g = GridGraph[10, 4]; h = DiscreteMath`OldCombinatorica`GridGraph[10, 4];
```

```
Timing[ VertexConnectivity[ g ]; ]
```

```
{7.485 Second, Null}
```

```
Timing[ DiscreteMath`OldCombinatorica`VertexConnectivity[ h ]; ]
```

```
{20.64 Second, Null}
```

VertexConnectivityGraph

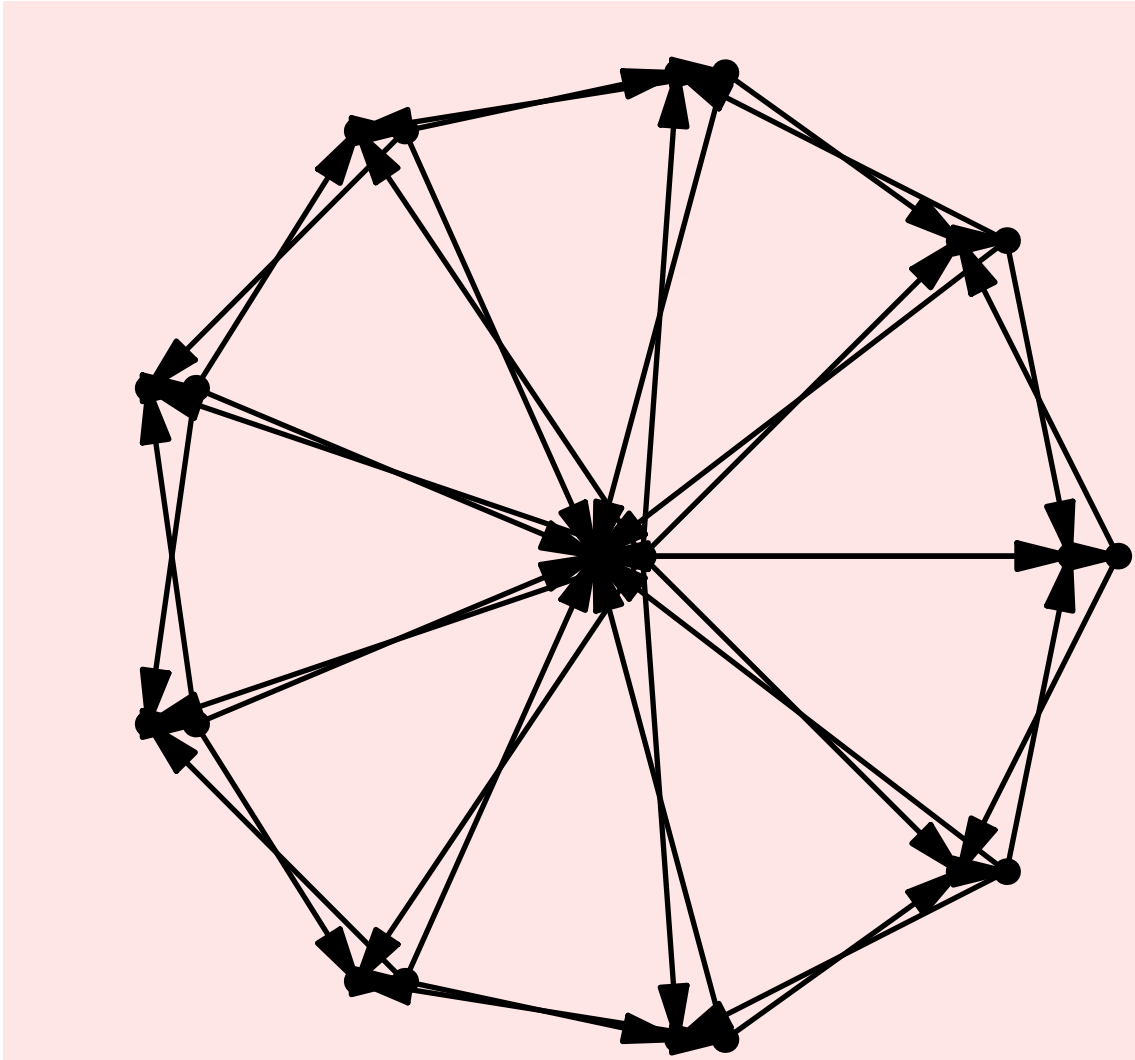
```
?VertexConnectivityGraph
```

```
VertexConnectivityGraph[g] returns a directed graph that
  contains an edge corresponding to each vertex in g and in which
  edge disjoint paths correspond to vertex disjoint paths in g.
```

NOTES

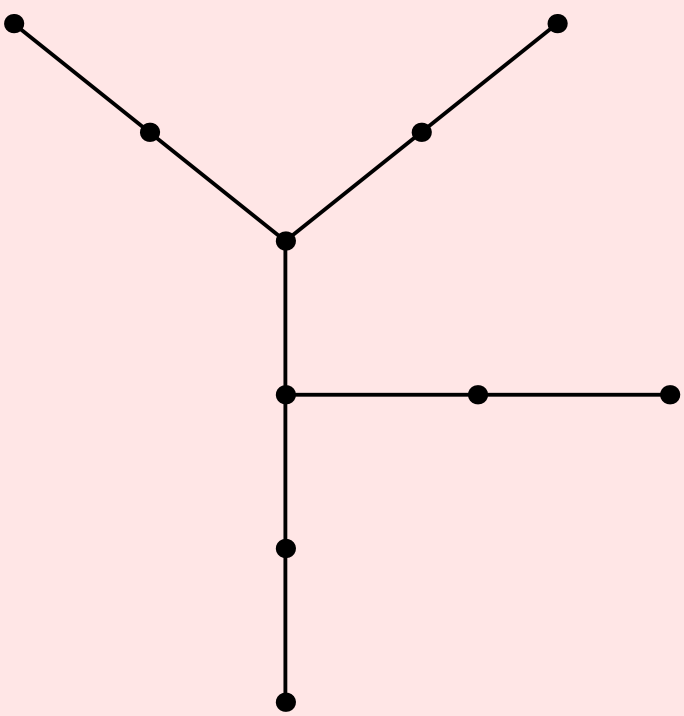
* This function was hidden in old Combinatorica. I think it produces interesting graphs and therefore have made the new function public. I have also tried to give the vertices a reasonable embedding that shows what transpired. The function produces directed graphs.

```
ShowGraph[g = VertexConnectivityGraph[Wheel[10]],
 VertexStyle -> Disc[0.025], ImageSize -> 400]
```



```
- Graphics -
```

```
ShowGraph[g = RandomTree[10]]
```

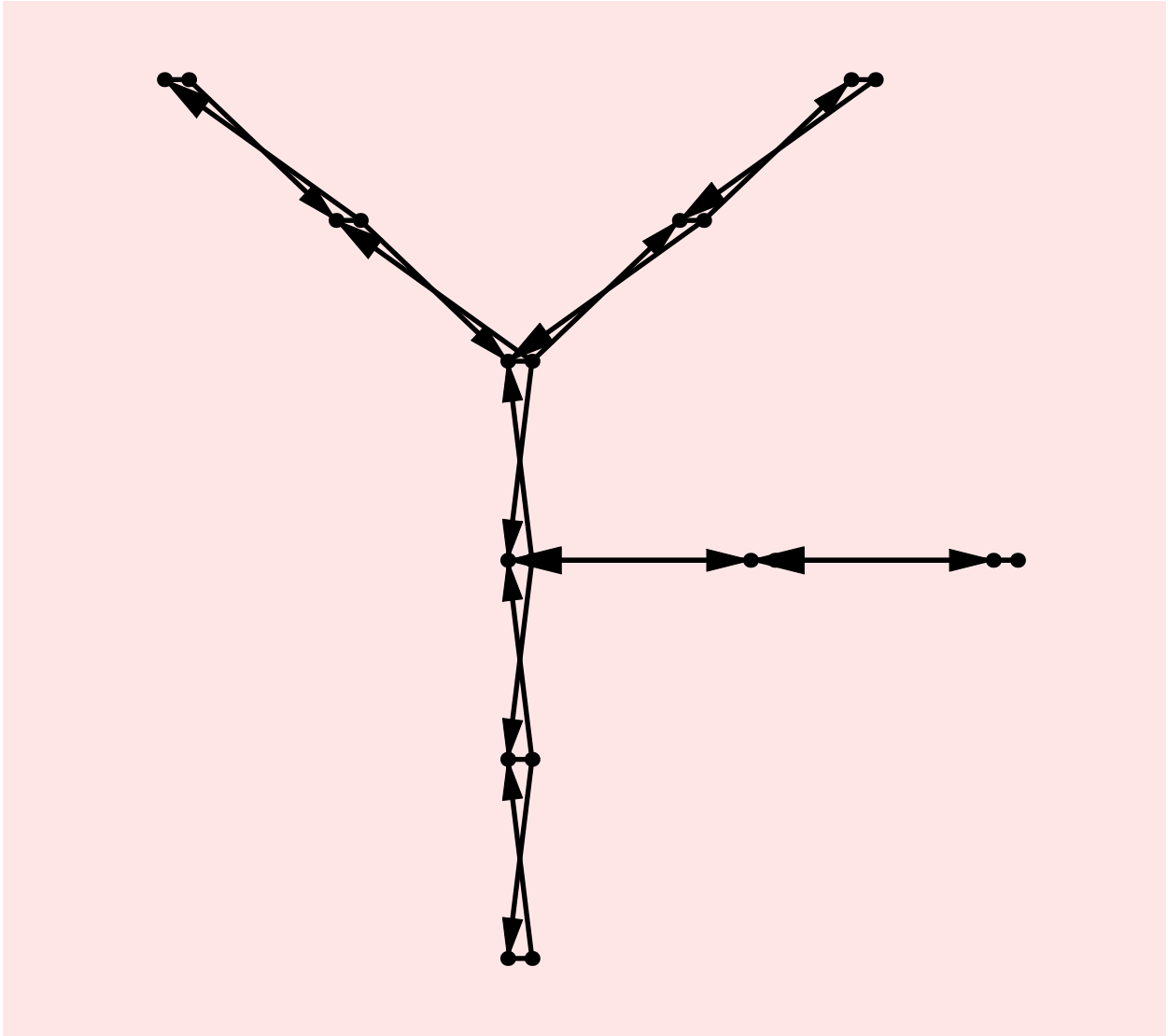


```
- Graphics -
```

NOTES

The following example also tries to show the use of relative headscaling for arrows. I thought this option would be very useful, but now I am skeptical. Look at the smallest edges −they seem not to have any arrow heads. That is only because their arrow heads are too small to be visible.

```
ShowGraph[VertexConnectivityGraph[g],
 HeadScaling -> Relative, ImageSize -> 400, VertexStyle -> Disc[0.015]]
```



```
- Graphics -
```

TIMING DISCUSSION

The above timing information shows that constructing the vertex connectivity graph takes virtually no time as compared to the computation of vertex connectivity itself: 0.02 s vc 26 s. So no significant improvement of VertexConnectivity will come from improving the construction of the vertex connectivity graph.

```
g = GridGraph[10, 4]; Timing[ VertexConnectivityGraph[g]; ]
```

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

TO DO

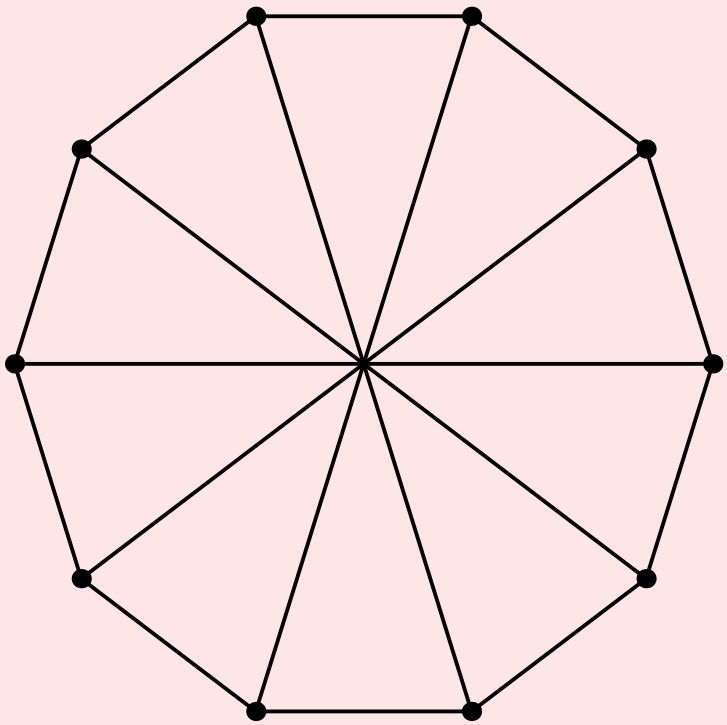
Both EdgeConnectivity and VertexConnectivity need to have options that will make the functions return a smallest edge−cut or smallest vertex−cutrespectively.

Harary

```
? Harary
```

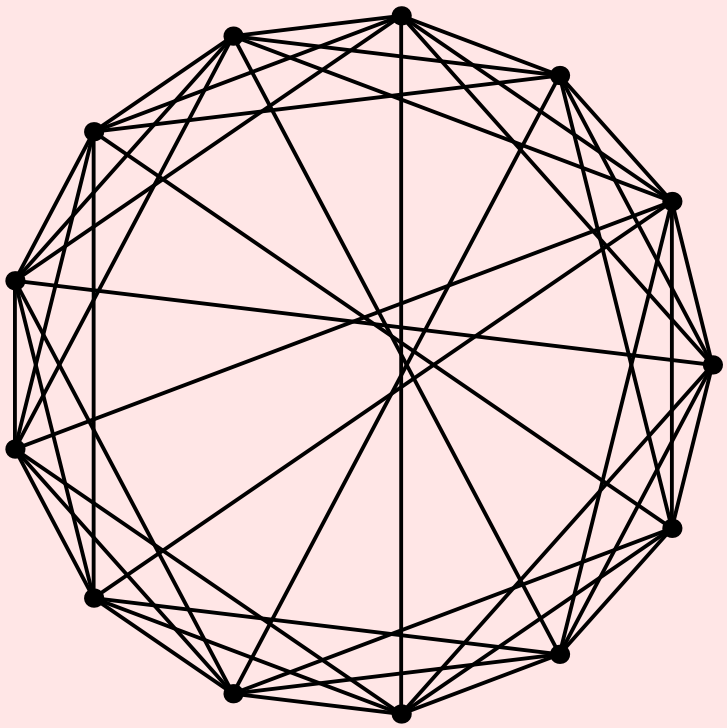
```
Harary[k, n] constructs the minimal k-connected graph on n vertices.
```

```
ShowGraph[Harary[3, 10]]
```



```
- Graphics -
```

```
ShowGraph[Harary[7, 13]]
```



```
- Graphics -
```

```
CompleteQ[Harary[8, 9]]
```

```
True
```

```
CompleteQ[Harary[10, 9]]
```

```
CompleteQ[Harary[10, 9]]
```

TIMING DISCUSSION

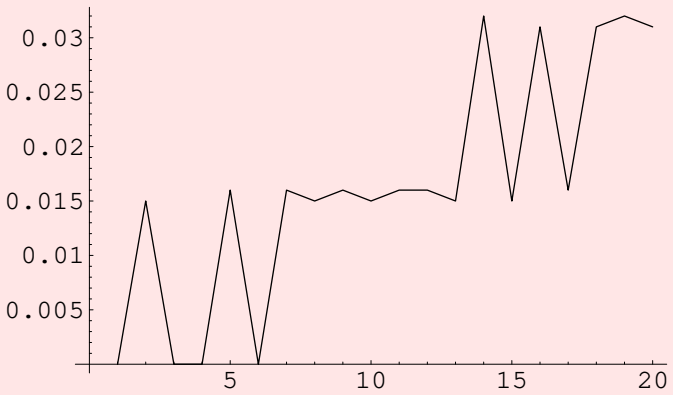
Harary[n, k] seems to be linear both in n and in k. I need to look at the code more carefully to determine if this is indeed true. I am surprised that it is linear in n.

As can be seen from the timing results below, construction of the Harary graph using the old version of the function is faster than using the new version. This is because Harary makes a call to CirculantGraph and CirculantGraph is extremely fast in the old version because it uses extremely convenient in–builtmatrix/vector operations. I don't think much can be done to improve the "new" version of Harary.

```
rt = Table[Timing[ Harary[5, i * 10];], {i, 20}]
```

```
{{0. Second, Null}, {0.015 Second, Null},
 {0. Second, Null}, {0. Second, Null}, {0.016 Second, Null},
 {0. Second, Null}, {0.016 Second, Null}, {0.015 Second, Null},
 {0.016 Second, Null}, {0.015 Second, Null}, {0.016 Second, Null},
 {0.016 Second, Null}, {0.015 Second, Null}, {0.032 Second, Null},
 {0.015 Second, Null}, {0.031 Second, Null}, {0.016 Second, Null},
 {0.031 Second, Null}, {0.032 Second, Null}, {0.031 Second, Null}}
```

```
ListPlot[ Map[#[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

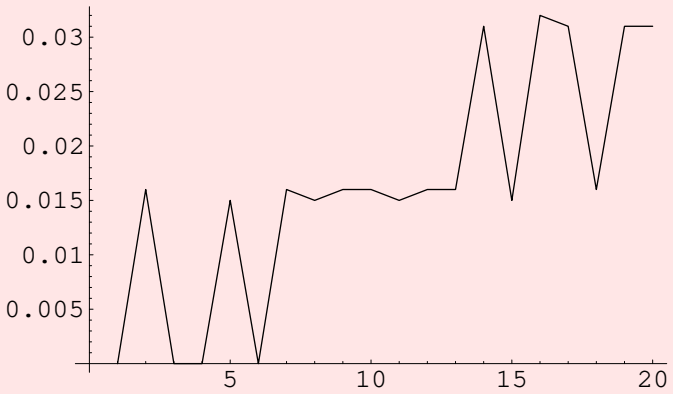
```
Table[Timing[ DiscreteMath`OldCombinatorica`Harary[5, i * 10];], {i, 20}]
```

```
{{0. Second, Null}, {0. Second, Null}, {0. Second, Null}, {0. Second, Null},
 {0. Second, Null}, {0. Second, Null}, {0. Second, Null}, {0. Second, Null},
 {0. Second, Null}, {0.016 Second, Null}, {0. Second, Null}, {0. Second, Null},
 {0. Second, Null}, {0.016 Second, Null}, {0. Second, Null}, {0. Second, Null},
 {0.015 Second, Null}, {0. Second, Null}, {0. Second, Null}, {0.016 Second, Null}}
```

```
rt = Table[Timing[ Harary[5, i * 10];], {i, 20}]
```

```
{{0. Second, Null}, {0.016 Second, Null},
 {0. Second, Null}, {0. Second, Null}, {0.015 Second, Null},
 {0. Second, Null}, {0.016 Second, Null}, {0.015 Second, Null},
 {0.016 Second, Null}, {0.016 Second, Null}, {0.015 Second, Null},
 {0.016 Second, Null}, {0.016 Second, Null}, {0.031 Second, Null},
 {0.015 Second, Null}, {0.032 Second, Null}, {0.031 Second, Null},
 {0.016 Second, Null}, {0.031 Second, Null}, {0.031 Second, Null}}
```

```
ListPlot[Map[#[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

BUG

The old Harary is producing an error message here. How come?

```
Table[Timing[DiscreteMath`OldCombinatorica`Harary[i*5, 10];], {i, 3, 20}]
```

IdenticalQ

```
?IdenticalQ
```

```
IdenticalQ[g, h] yields True if graphs
  g and h have identical edge lists, even though the
  associated graphics information need not be the same.
```

```
g = CompleteGraph[10];
```

```
IdenticalQ[g, CompleteGraph[10]]
```

```
True
```

```
IdenticalQ[g, Wheel[10]]
```

```
False
```

```
g = AddEdges[g, {{{1, 2}}}];
```

```
IdenticalQ[g, CompleteGraph[10]]
```

```
False
```

```
IdenticalQ[CirculantGraph[10, Range[5]], CompleteGraph[10]]
```

```
True
```

```
IdenticalQ[
 RemoveMultipleEdges[CirculantGraph[10, Range[5]]], CompleteGraph[10]]
```

```
True
```

```
IdenticalQ[GraphDifference[Wheel[11], AddVertices[Cycle[10], 1 ]], Star[11]]
```

```
True
```

```
g = GridGraph[100, 100];
```

```
Timing[ IdenticalQ[g, g];]
```

```
{0.14 Second, Null}
```

TIMING DISCUSSION

The "new" version of IdenticalQ is quite fast −10, 000 vertex−graphsin less than a second. Of course the old version of this function is also extremely fast since it involves comparing the two adjacency matrices.

IsomorphismQ

```
?IsomorphismQ
```

```
IsomorphismQ[g, h, p] tests if permutation
  p defines an isomorphism between graphs g and h.
```

```
IsomorphismQ[CompleteGraph[10], CompleteGraph[10], RandomPermutation[10]]
```

```
True
```

```
tf = Map[IsomorphismQ[Cycle[4], Cycle[4], #] &, Permutations[Range[4]]]
```

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

```
Length[Select[tf, (# == True) &]]
```

```
8
```

NOTES

* The automorphism group of a 4−cycleis an order−8subgroup of the symmetric group S_4. This is basically the dihedral group on 4 elements.

TIMING DISCUSSION

The new implementation of this function seems to be quadratic in running time. This is because InducePermuteSubgraph is quadratic.This should be speeded up.

TO DO

Speed up the InducePermuteSubgraph function.

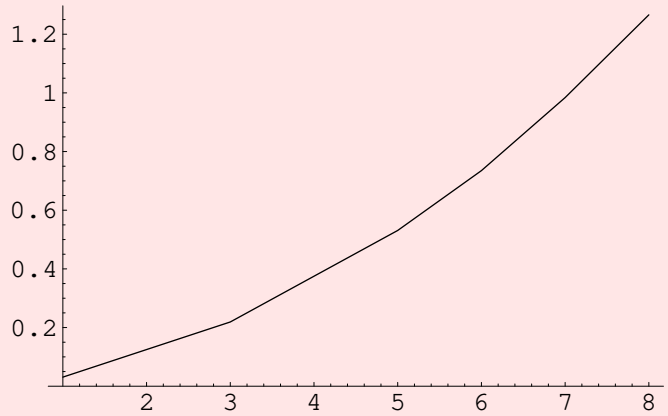
The old implementation of function is much faster than the new implementation because the old InduceSubgraph is lightning fast!

```
gt = Table[GridGraph[20, 10 i], {i, 8}];
pt = Table[RandomPermutation[200 i], {i, 8}];
```

```
rt = Table[Timing[ IsomorphismQ[gt[[i]] , gt[[i]], pt[[i]] ]; ], {i, 8}]
```

```
{{0.031 Second, Null}, {0.125 Second, Null},
 {0.219 Second, Null}, {0.375 Second, Null}, {0.531 Second, Null},
 {0.735 Second, Null}, {0.984 Second, Null}, {1.266 Second, Null}}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
ht = Table[ DiscreteMath`OldCombinatorica`GridGraph[20, 10 i], {i, 8}];
```

```
rt = Table[ Timing[ DiscreteMath`OldCombinatorica`IsomorphismQ[
     gt[[i]], gt[[i]], pt[[i]] ]; ], {i, 8}]
```

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

Isomorphism

```
? Isomorphism
```

```
Isomorphism[g, h] gives an isomorphism between graphs g and h
  if one exists. Isomorphism[g, h, All] gives all isomorphisms
  between graphs g and h. Isomorphism[g] gives the automorphism
  group of g. This function takes an option Invariants -> {f1,
  f2, ...}, where f1, f2, ... are functions that are used to
  compute vertex invariants. These functions are used in the order
  in which they are specified. The default value of Invariants
  is {DegreesOf2Neighborhood, NumberOf2Paths, Distances}.
```

```
Isomorphism[Wheel[10], CompleteGraph[10]]
```

```
{}
```

```
Isomorphism[Wheel[5], Wheel[5], All]
```

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

```
Isomorphism[Star[5], Star[5], All]
```

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

```
Isomorphism[Wheel[10], Star[10]]
```

```
{}
```

```
Isomorphism[Cycle[10], Cycle[10], All]
```

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

NOTES

* The above is the automorphism group of a 10–cycle.This is isomorphic to the dihedral group on 10 symbols.

TIMING DISCUSSION

The isomorphism function is extremely slow –the newer version of the function seems slower than the older (already slow) function. This function, needs major repair. It would be very nice to add a variety of heuristics to this function that help it prune the search space quickly. Clearly, Backtrack, as it is does not suffice. Brendan McKay's Nauty specializes in graph isomorphism –I should look that up and see how it does what it does.

```
g = Table[Cycle[i], {i, 10, 20, 2}]; h = Table[Cycle[i], {i, 10, 20, 2}];
```

```
Table[Timing[ Isomorphism[g[[i]], g[[i]], All];], {i, 6}]
```

```
$Aborted
```

```
Table[Timing[
  DiscreteMath`OldCombinatorica`Isomorphism[h[[i]], h[[i]], All]; ], {i, 6}]
```

```
{{0. Second, Null}, {0.015 Second, Null}, {0. Second, Null},
 {0. Second, Null}, {0. Second, Null}, {0. Second, Null}}
```

NOTES

* Now the new Isomorphism also yields the automorphism group if called with a single graph.

```
Isomorphism[Cycle[5]]
```

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

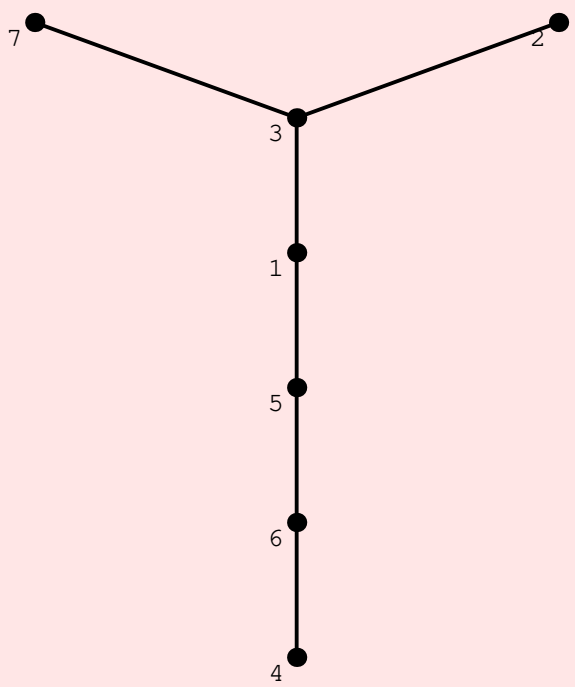
```
Isomorphism[Path[5]]
```

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

```
Isomorphism[t = RandomTree[7]]
```

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

```
ShowGraph[t, VertexNumber -> On, PlotRange -> Large[.1]]
```




```
- Graphics -
```

IsomorphicQ

```
?IsomorphicQ
```

```
IsomorphicQ[g, h] yields True if graphs g and h are isomorphic.
  This function takes an option Invariants -> {f1, f2, ...},
  where f1, f2, ... are functions that are used to compute
  vertex invariants. These functions are used in the order in
  which they are specified. The default value of Invariants
  is {DegreesOf2Neighborhood, NumberOf2Paths, Distances}.
```

```
IsomorphicQ[Wheel[10], Wheel[10]]
```

```
True
```

```
IsomorphicQ[CompleteGraph[8], GraphJoin[CompleteGraph[4], CompleteGraph[4]]]
```

```
True
```

```
IsomorphicQ[ RandomTree[20], CompleteGraph[20]]
```

```
False
```

Equivalences

```
?Equivalences
```

```
Equivalences[g, h] lists the vertex equivalence classes between graphs g
  and h defined by their vertex degrees. Equivalences[g] lists the
  vertex equivalences for graph g defined by the vertex degrees.
  Equivalences[g, h, f1, f2, ...] and Equivalences[g, f1, f2, ...]
  can also be used, where f1, f2, ... are functions that compute
  other vertex invariants. It is expected that for each function
  fi, the call fi[g, v] returns the corresponding invariant at
  vertex v in graph g. The functions f1, f2, ... are evaluated
  in order, and the evaluation stops either when all functions
  have been evaluated or when an empty equivalence class is found.
```

NOTES

* The example below shows that the first 9 vertices in Wheel[10], each of which has degree 3, cannot be mapped on to any vertex in CompleteGraph[10], while the 10th vertex in Wheel[10], which has degree 9, can be mapped on to any vertex in CompleteGraph[10].

```
Equivalences[Wheel[10], CompleteGraph[10]]
```

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

```
Equivalences[Path[10], CompleteGraph[10]]
```

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

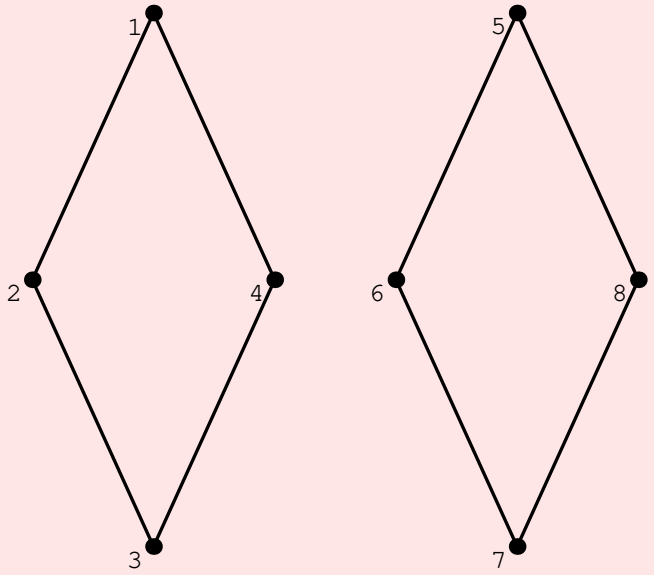
```
Equivalences[Path[10], Path[10]]
```

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

NOTES

Below I construct two non−isomorphic8−vertex3−regulargraphs. The first of these these two graphs, has several triangles while the second graph is Hypercube[3], which is bipartite. Note that since Equivalences[g, h] pays attention only to vertex degrees it ends up concluding that every vertex in the first graph can be mapped on to every vertex in the second graph. Looking at the adjacency matrix of the squares of these graphs quickly resolves the confusion and

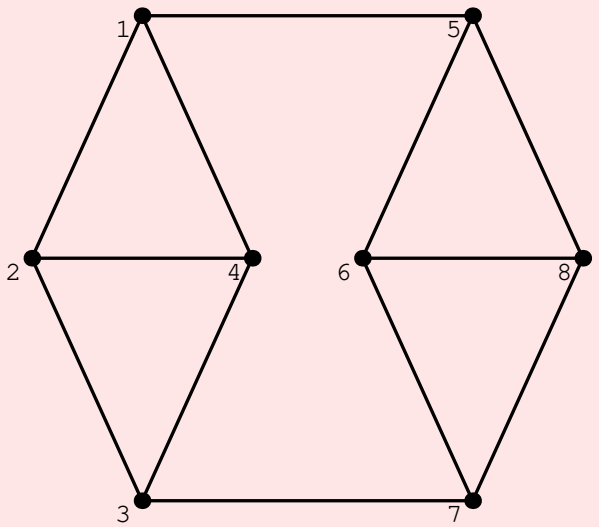
```
g = GraphUnion[ Cycle[4], Cycle[4]]; ShowGraph[g, VertexNumber -> On]
```




```
- Graphics -
```

```
g = AddEdges[g, { {{1, 5}}, {{2, 4}}, {{6, 8}}, {{3, 7}} }];
```

```
ShowGraph[g, VertexNumber -> On, PlotRange -> Large[0.1]]
```

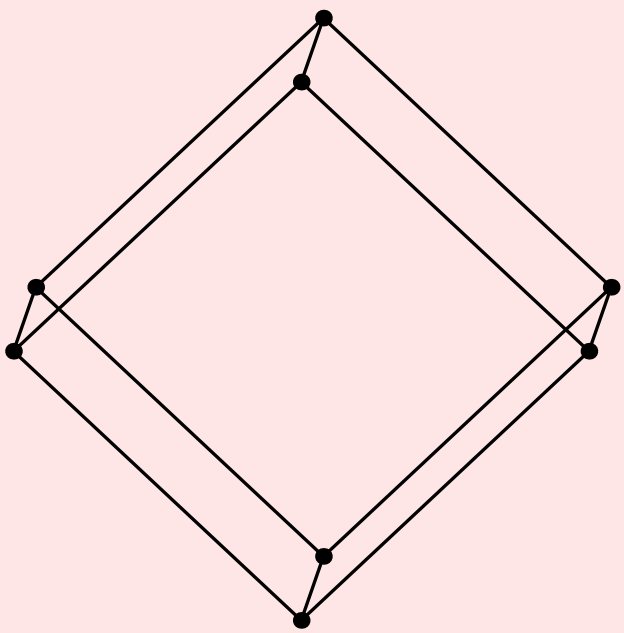



```
- Graphics -
```

```
DegreeSequence[ g ]
```

```
{3, 3, 3, 3, 3, 3, 3, 3}
```

```
ShowGraph[h = Hypercube[ 3 ] ]
```



```
- Graphics -
```

```
DegreeSequence[ h ]
```

```
{3, 3, 3, 3, 3, 3, 3, 3}
```

```
Equivalences[g, h ]
```

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

```
Isomorphism[g, h, All]
```

```
{}
```

```
Equivalences[ GraphPower[g, 2], GraphPower[h, 2]]
```

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

NOTES

* This is simply the beginnings of how I think Isomorphism can be improved. I think that using AllPairsShortestPaths is probably too costly a method to determine equivalences. It would definitely be faster to use the number of paths matrix for paths of some small constant length. This matrix can be computed by simple matrix multiplication, which will make the computation relatively fast. Also, the user should be allowed to play with Equivalences so that using different methods, the equivalences can be made more fine. Here is an example.

```
gm = Path[10]; gm2 = GraphPower[gm, 2];
```

```
Equivalences[gm2, gm2]
```

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

NOTE

* Notice that since we used the 2nd power of the adjacency matrix, we have finer equivalence classes. Similarly, the situation improves with the third power of the adjacency matrix. Finally, when we get to the 5th power of adjacency matrix, it is clear that there are only two candidate permutations that we need to try: identity and the path "flip". In this case, both are isomorphisms.

```
Equivalences[gm3 = GraphPower[gm, 3], gm3]
```

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

```
Equivalences[gm5 = GraphPower[gm, 5], gm5]
```

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

```
Equivalences[Wheel[5]]
```

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

```
Equivalences[CompleteGraph[4]]
```

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

Automorphisms

```
?Automorphisms
```

```
Automorphisms[g] gives the automorphism group of the graph g.
```

```
Automorphisms[Star[4]]
```

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

SelfComplementaryQ

```
?SelfComplementaryQ
```

```
SelfComplementaryQ[g] yields True if graph g is self-
  complementary, meaning it is isomorphic to its complement.
```

```
SelfComplementaryQ[Path[4]]
```

```
True
```

```
SelfComplementaryQ[Cycle[5]]
```

```
True
```

```
SelfComplementaryQ[Path[5]]
```

```
False
```

FindCycle

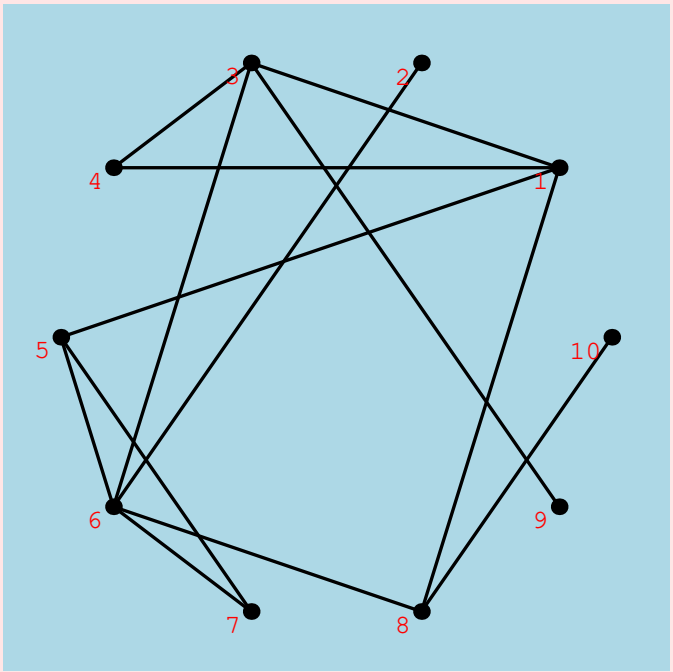
```
?FindCycle
```

```
FindCycle[g] finds a list of vertices that define a cycle in graph g.
```

NOTES

* FindCycle needs to have an edge option that will cause the function to return the edges in a cycle. This option can then be carried over to ExtractCycles.

```
ShowGraph[t = RandomGraph[10, .4], VertexNumber -> On,
 VertexNumberColor -> Red, PlotRange -> Large[0.1], Background -> LightBlue]
```

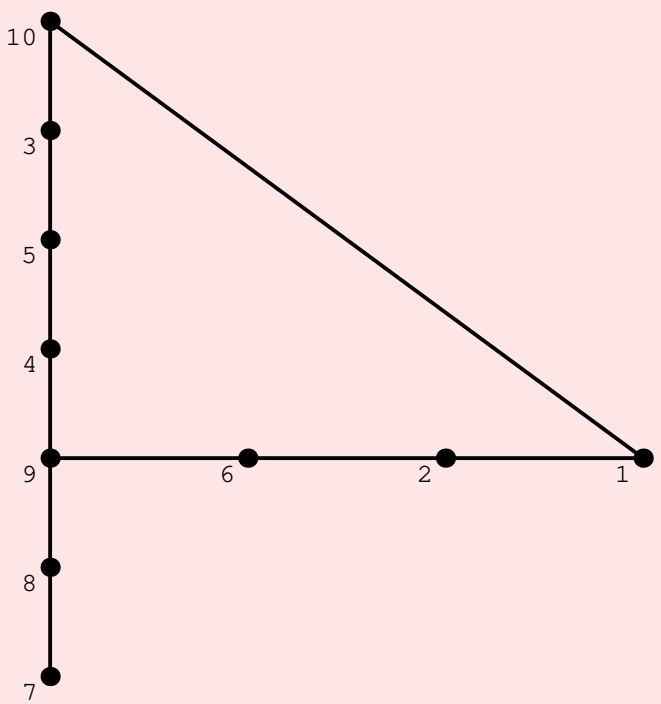



```
- Graphics -
```

```
FindCycle[t]
```

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

```
ShowGraph[t = AddEdges[RandomTree[10], {{{1, 10}}}],
 VertexNumber -> On, PlotRange -> Large[0.05]]
```




```
- Graphics -
```

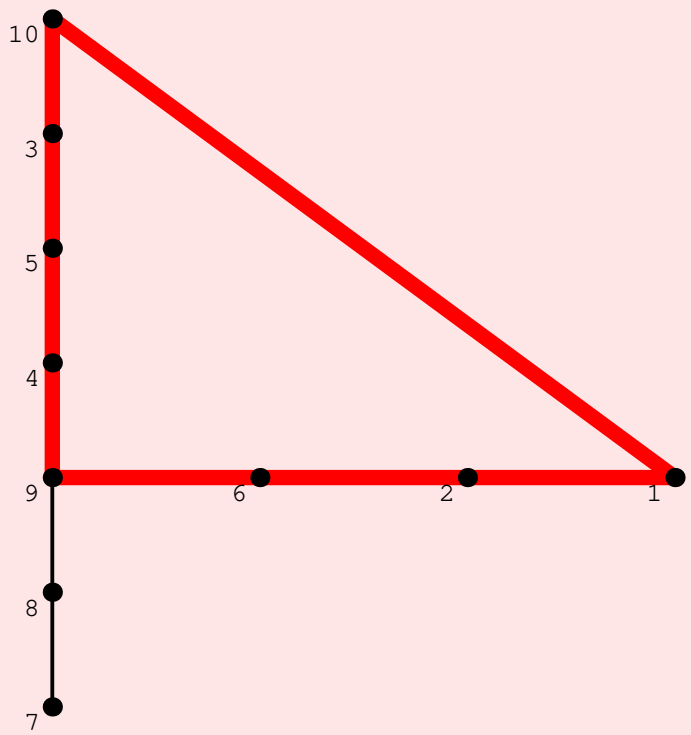
```
FindCycle[t]
```

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

```
HighlightCycle[t_Graph] := Highlight[t, c = FindCycle[t];
   {Table[Sort[{c[[i]], c[[i + 1]]}], {i, Length[c] - 1}]},
   HighlightedEdgeColors -> {Red}] /; UndirectedQ[t]
```

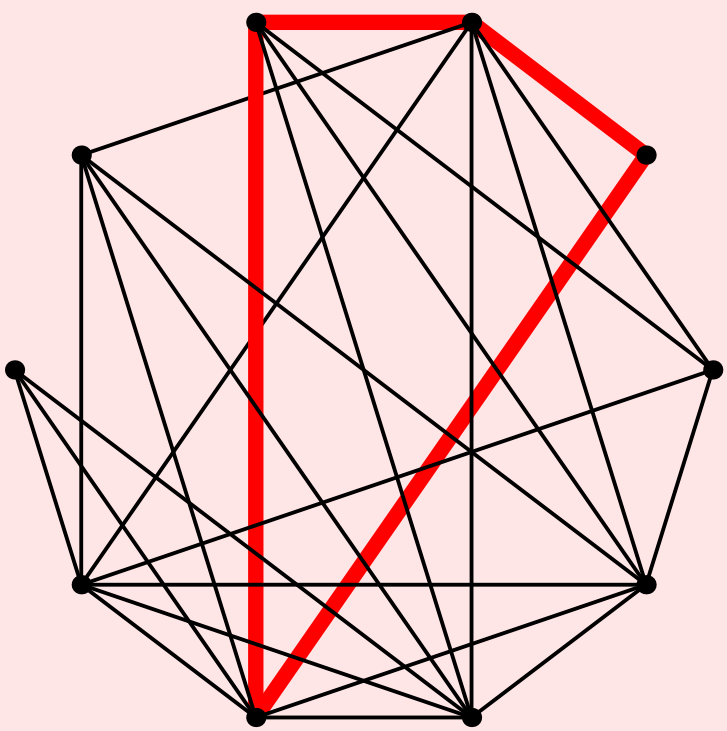
```
HighlightCycle[t_Graph] := Highlight[t,
  c = FindCycle[t]; {Table[{c[[i]], c[[i + 1]]}, {i, Length[c] - 1}]},
  HighlightedEdgeColors -> {Red}]
```

```
ShowGraph[HighlightCycle[t], VertexNumber -> On]
```



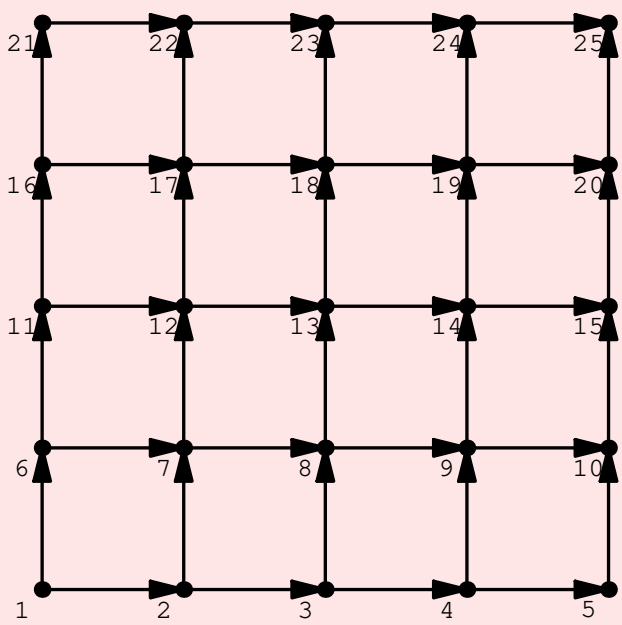

```
- Graphics -
```

```
ShowGraph[HighlightCycle[RandomGraph[10, .4]]]
```



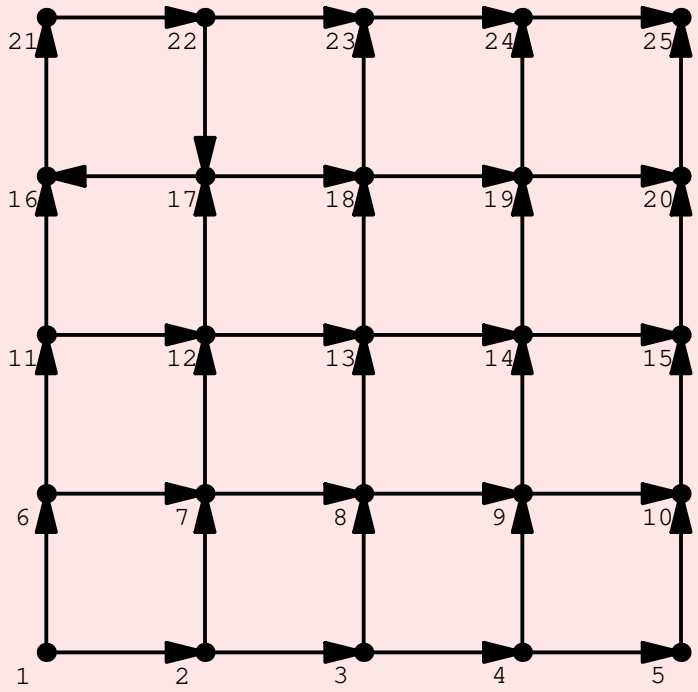
```
- Graphics -
```

```
ShowGraph[t = SetGraphOptions[GridGraph[5, 5], EdgeDirection -> On,
   VertexNumber -> Text[{-0.03, -0.03}]], PlotRange -> Large[0.07]]
```



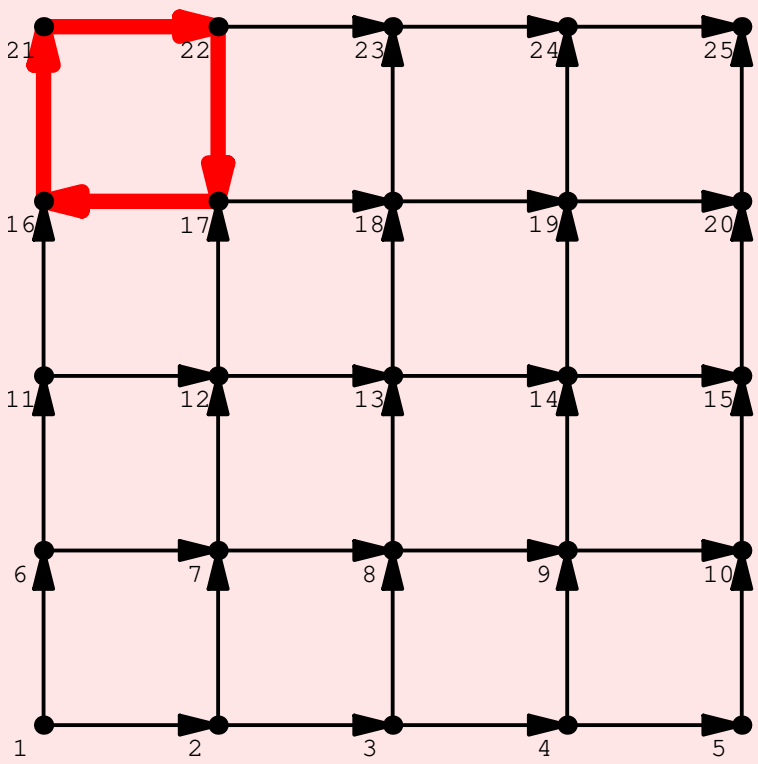

```
- Graphics -
```

```
ShowGraph[t =
  AddEdges[DeleteEdges[t, {{16, 17}, {17, 22}}], {{{17, 16}}, {{22, 17}}}],
 VertexNumber -> On, PlotRange -> Large[0.1]]
```




```
- Graphics -
```

```
ShowGraph[HighlightCycle[t]]
```




```
- Graphics -
```

NOTES

* FindCycle can deal correctly with multiple edges. It reports 2−cyclescorrectly. In fact, it reports self−loopsalso correctly.

```
t = AddEdges[Path[4], {{{1, 2}}}];
```

```
Edges[t]
```

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

```
FindCycle[t]
```

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

```
t = AddEdges[ Path[4], { {{1, 1}} } ];
```

```
FindCycle[ t ]
```

```
{1, 1}
```

TIMING DISCUSSION

The plot of the table of timings for FindCyle on random graphs is quite strange –partly caused by the fact that the graphs are random and partly caused by the fact that the function stops as soon as it finds a cycle.

The plot of FindCycle timings for grid graphs is shown below. This is linear, but this is just because the function quickly finds 4–cyclesin each of these graphs. The asymptotic running time of the function is definitely quadratic and this needs to be fixed.

The new version of FindCyle is faster than the old version, but the new version could be improved some more.

TO DO

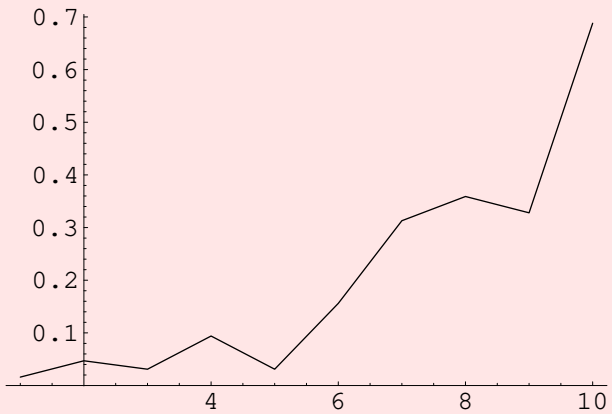
Speed up FindCycle.

```
gt = Table[ RandomGraph[50 i, N[1 / (50 i)]], {i, 10}];
```

```
rt = Table[ Timing[ FindCycle[gt[[i]] ];], {i, 10}]
```

```
{{0.016 Second, Null}, {0.047 Second, Null},
 {0.031 Second, Null}, {0.094 Second, Null},
 {0.031 Second, Null}, {0.156 Second, Null}, {0.313 Second, Null},
 {0.359 Second, Null}, {0.328 Second, Null}, {0.688 Second, Null}}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
gt = Table[ GridGraph[20, 10 i], {i, 10}];
```

```
ct = Table[0, {10}];
```

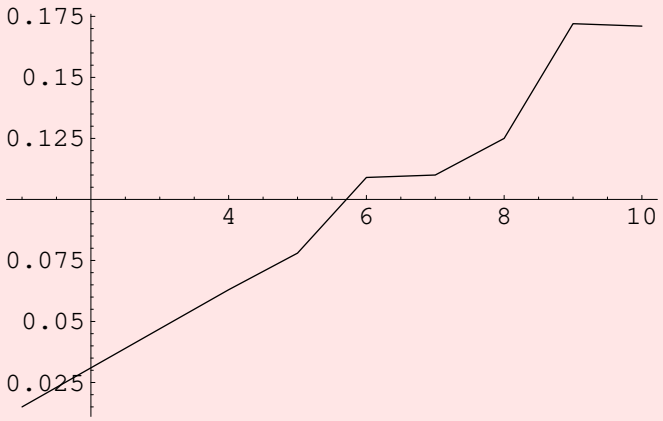
```
rt = Table[ Timing[ct[[i]] = FindCycle[gt[[i]] ];], {i, 10}]
```

```
{{0.015 Second, Null}, {0.031 Second, Null},
 {0.047 Second, Null}, {0.063 Second, Null},
 {0.078 Second, Null}, {0.109 Second, Null}, {0.11 Second, Null},
 {0.125 Second, Null}, {0.172 Second, Null}, {0.171 Second, Null}}
```

```
ct
```

```
{{39, 19, 20, 40, 39}, {39, 19, 20, 40, 39},
 {39, 19, 20, 40, 39}, {39, 19, 20, 40, 39},
 {39, 19, 20, 40, 39}, {39, 19, 20, 40, 39}, {39, 19, 20, 40, 39},
 {39, 19, 20, 40, 39}, {39, 19, 20, 40, 39}, {39, 19, 20, 40, 39}}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
ht =
  Table[DiscreteMath`OldCombinatorica`RandomGraph[50 i, N[1 / (50 i)]], {i, 5}]
```

```
rt = Table[Timing[ DiscreteMath`OldCombinatorica`FindCycle[ht[[i]] ]; ], {i, 5}
```

```
{{0.016 Second, Null}, {0.062 Second, Null},
 {0.188 Second, Null}, {0.281 Second, Null}, {0.797 Second, Null}}
```

AcyclicQ

```
?AcyclicQ
```

```
AcyclicQ[g] yields True if graph g is acyclic.
```

```
AcyclicQ[RandomTree[100]]
```

```
True
```

```
AcyclicQ[GridGraph[10, 10]]
```

```
False
```

```
AcyclicQ[SetGraphOptions[GridGraph[10, 10], EdgeDirection -> On]]
```

```
True
```

TreeQ

```
?TreeQ
```

```
TreeQ[g] yields True if graph g is a tree.
```

```
TreeQ[Star[100]]
```

```
True
```

```
$RecursionLimit = 10000;
```

```
TreeQ[Path[100]]
```

```
True
```

```
TreeQ[RandomTree[10]]
```

```
True
```

```
TreeQ[CompleteGraph[10]]
```

```
False
```

```
TreeQ[Wheel[10]]
```

```
False
```

ExtractCycles

```
?ExtractCycles
```

```
ExtractCycles[g] gives a maximal list of edge-disjoint cycles in graph g.
```

```
ExtractCycles[Wheel[10]]
```

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

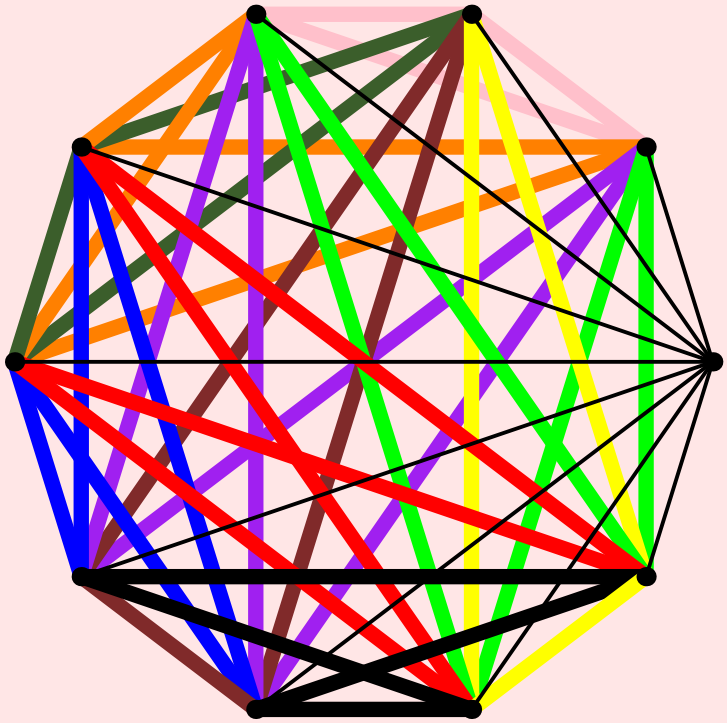
```
cl = ExtractCycles[CompleteGraph[10]]
```

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

```
cl = Map[Map[Sort, Partition[#, 2, 1]] &, cl]
```

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

```
ShowGraph[Highlight[CompleteGraph[10], cl]]
```



```
- Graphics -
```

TIMING DISCUSSION

As I discovered when I timed OrientGraph, the ExtractCycles function is extremely and unnecessarily slow. This function seems amenable to significant speedup. Here is a timing experiment. Though the new version seems to be roughly 3–4 times faster than the old function, it could probably be much faster.

```
g = Table[GridGraph[i * 5, 5], {i, 5}];
```

```
Table[ Timing[ ExtractCycles[g[[i]]]; ], {i, 5}]
```

```
{{0.078 Second, Null}, {0.266 Second, Null},
 {0.594 Second, Null}, {1.125 Second, Null}, {1.797 Second, Null}}
```

```
h = Table[DiscreteMath`OldCombinatorica`GridGraph[i * 5, 5], {i, 5}];
```

```
Table[ Timing[ DiscreteMath`OldCombinatorica`ExtractCycles[h[[i]]]; ], {i, 5}]
```

```
{{0.078 Second, Null}, {0.438 Second, Null},
 {1.203 Second, Null}, {2.672 Second, Null}, {4.969 Second, Null}}
```

DeleteCycle

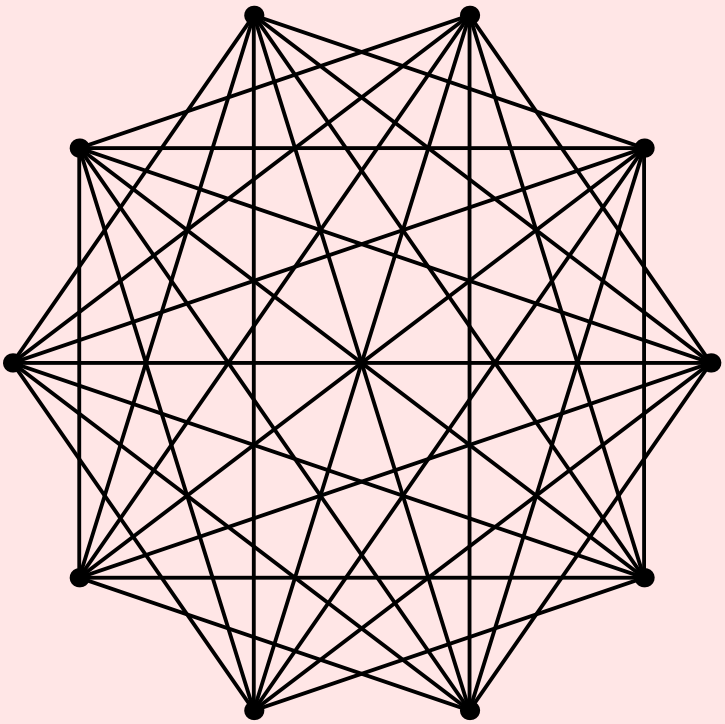
```
?DeleteCycle
```

```
DeleteCycle[g, c] deletes a simple cycle c from graph g. c is specified
  as a sequence of vertices in which the first and last vertex are
  identical. g can be directed or undirected. If g does not contain
  c, it is returned unchanged, otherwise g is returned with c deleted.
```

NOTE

* This function should probably be able to delete several cycles in one shot.

```
ShowGraph[DeleteCycle[CompleteGraph[10], {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1}]]
```



```
- Graphics -
```

```
IdenticalQ[DeleteCycle[Wheel[10], {1, 2, 3, 4, 5, 6, 7, 8, 9, 1}], Star[10]]
```

```
True
```

TIMING DISCUSSION

Both versions of DeleteCycle are pretty fast and there is not much scope for improvement here.

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

```
Timing[ DeleteCycle[g, {1, 2, 3, 33, 32, 31, 1}]; ]
```

```
{0.094 Second, Null}
```

```
h = DiscreteMath`OldCombinatorica`GridGraph[30, 30];
```

```
Timing[
 DiscreteMath`OldCombinatorica`DeleteCycle[g, {1, 2, 3, 33, 32, 31, 1}]; ]
```

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

Girth

```
?Girth
```

```
Girth[g] gives the length of the shortest cycle in a simple graph g.
```

NOTES

* The new Girth should probably return a cycle that realizes thegirth. It might have been fun to have a construction of a graph with arbitrarily large girth AND arbitrarily large chromatic number. Nesetril and Rodl supposedly have a construction that is not too complicated.

```
Girth[Wheel[10]]
```

```
3
```

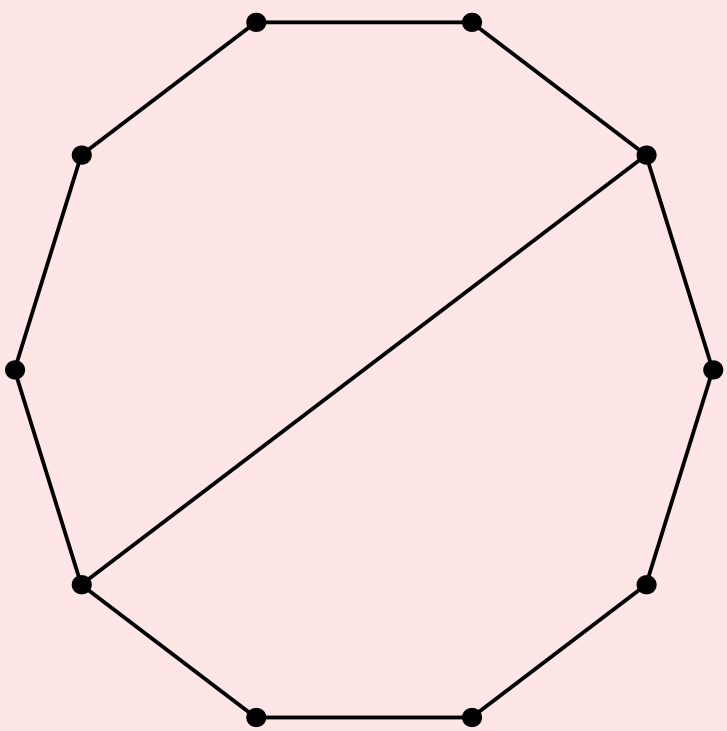
```
Girth[GridGraph[5, 5]]
```

```
4
```

```
Girth[Cycle[10]]
```

```
10
```

```
ShowGraph[t = AddEdges[Cycle[10], {{{1, 6}}}]]
```



```
- Graphics -
```

```
Girth[t]
```

```
6
```

```
Girth[CompleteGraph[4, 4]]
```

```
4
```

```
Girth[RandomTree[10]]
```

```
∞
```

TIMING DISCUSSION

I need to look at the code of Girth to figure out what its asymptotic running time is. The plot below indicates that it is

superlinear. I suspect it's running time is cubic.

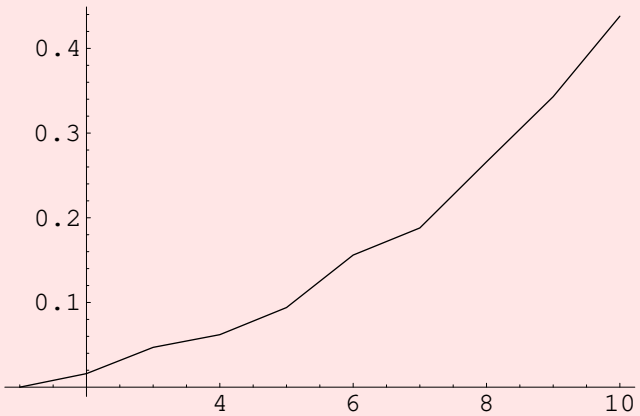
The new version of Girth is a shade faster than the older version of Girth. Can it be speeded up significantly?

```
gt = Table[RandomGraph[i * 10, .4], {i, 1, 10}];
```

```
rt = Table [ Timing[ Girth[gt[[i]]]; ], {i, 10}]
```

```
{{0. Second, Null}, {0.016 Second, Null},
 {0.047 Second, Null}, {0.062 Second, Null},
 {0.094 Second, Null}, {0.156 Second, Null}, {0.188 Second, Null},
 {0.266 Second, Null}, {0.343 Second, Null}, {0.438 Second, Null}}
```

```
ListPlot[ Map[ #[[1, 1]] &, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
ht = Table[DiscreteMath`OldCombinatorica`RandomGraph[i * 10, .4], {i, 1, 10}];
```

```
rt = Table [ Timing[DiscreteMath`OldCombinatorica`Girth[ht[[i]]]; ], {i, 10}]
```

```
{{0. Second, Null}, {0.031 Second, Null},
 {0.031 Second, Null}, {0.063 Second, Null},
 {0.125 Second, Null}, {0.156 Second, Null}, {0.219 Second, Null},
 {0.281 Second, Null}, {0.375 Second, Null}, {0.438 Second, Null}}
```

OutDegree

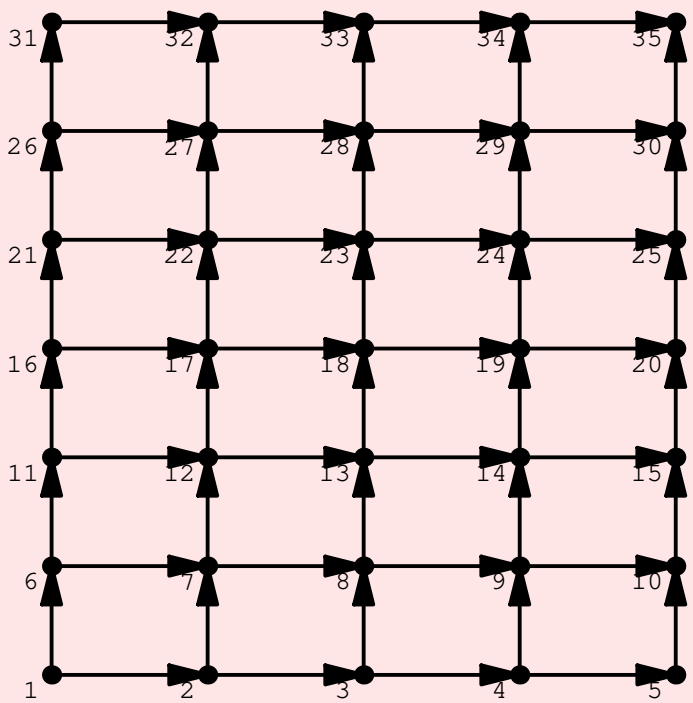
```
? OutDegree
```

```
OutDegree[g, n] returns the out-degree of vertex
  n in directed graph g. OutDegree[g] returns the sequence
  of out-degrees of the vertices in directed graph g.
```

```
OutDegree[t = SetGraphOptions[GridGraph[5, 7], EdgeDirection -> On]]
```

```
{2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1,
 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 1, 1, 1, 1, 0}
```

```
ShowGraph[t, VertexNumber -> On, PlotRange -> Large[0.07]]
```




```
- Graphics -
```

```
OutDegree[
 SetGraphOptions[GraphUnion[10, CompleteGraph[1]], EdgeDirection -> On]]
```

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

InDegree

```
? InDegree
```

```
InDegree[g, n] returns the in-degree of vertex
  n in directed graph g. InDegree[g] returns the sequence
  of in-degrees of the vertices in directed graph g.
```

```
InDegree[SetGraphOptions[GridGraph[5, 5], EdgeDirection -> On]]
```

```
{0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2}
```

ReverseEdges

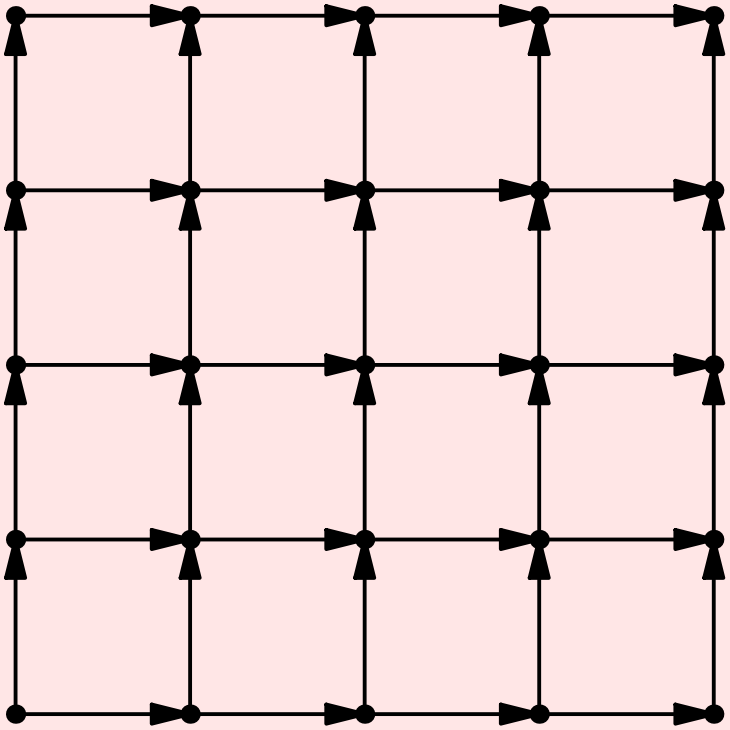
```
? ReverseEdges
```

```
ReverseEdges[g] flips the directions of all edges in a directed graph.
```

NOTES

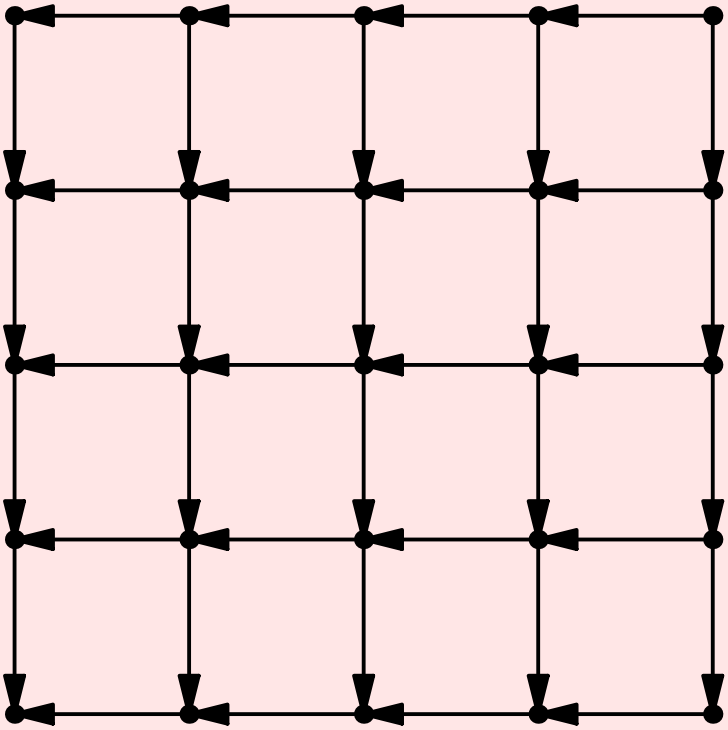
* The new ReverseEdges has been created to help in writing the new InDegree function. This plays the same role that TransposeGraph played in the earlier implementation. Now I have gotten rid of TransposeGraph. Two questions I have are (1) Should the new ReverseEdges be public? (2) Should the new ReverseEdges be more general in the sense that any subset of edges should be reversible?

```
ShowGraph[t = SetGraphOptions[GridGraph[5, 5], EdgeDirection -> On]]
```



```
- Graphics -
```

```
ShowGraph[ReverseEdges[t]]
```



```
- Graphics -
```

EulerianQ

```
?EulerianQ
```

```
EulerianQ[g] yields True if the graph g is Eulerian, meaning
  there exists a tour which includes each edge exactly once.
```

```
EulerianQ[CompleteGraph[9]]
```

```
True
```

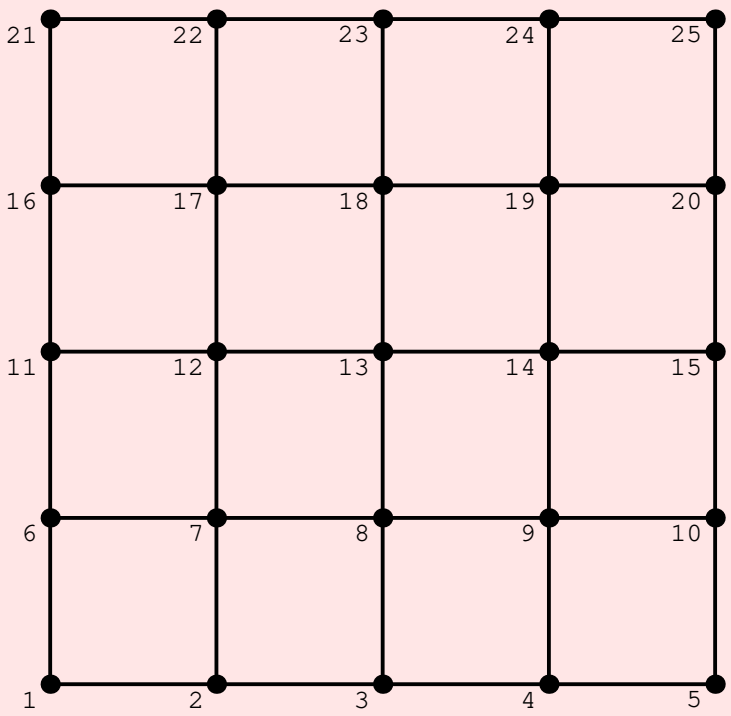
```
EulerianQ[Cycle[10]]
```

```
True
```

```
EulerianQ[t = GridGraph[5, 5]]
```

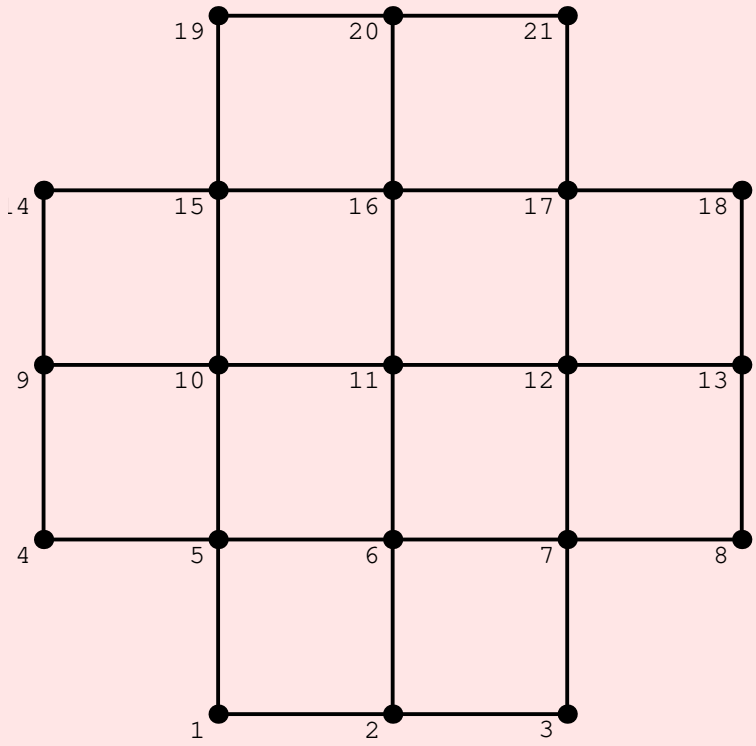
```
False
```

```
ShowGraph[t, VertexNumber -> On, PlotRange -> Large[0.05]]
```




```
- Graphics -
```

```
ShowGraph[t = DeleteVertices[t, {1, 5, 21, 25}], VertexNumber -> On]
```

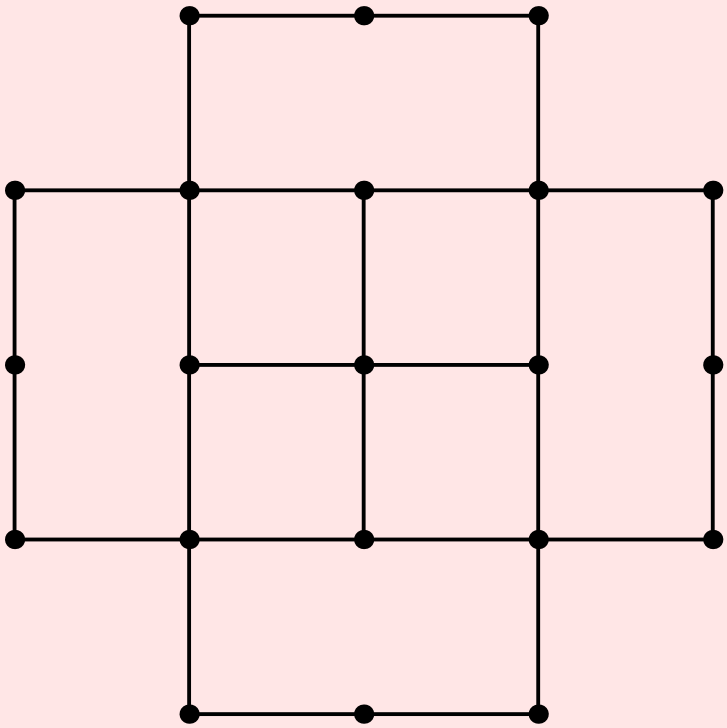



```
- Graphics -
```

```
EulerianQ[t]
```

```
False
```

```
ShowGraph[t = DeleteEdges[t, {{9, 10}, {16, 20}, {12, 13}, {2, 6}}]]
```



```
- Graphics -
```

```
EulerianQ[t]
```

```
False
```

```
$RecursionLimit = 100000000000
```

```
100000000000
```

```
g = GridGraph[70, 70];
```

```
Timing[ EulerianQ[g]; ]
```

```
{15.391 Second, Null}
```

```
Timing[ ConnectedQ[g]; ]
```

```
{15.141 Second, Null}
```

The new EulerianQ is reatively slow because of new ConnectedQ, as can be seen from the above example. When I replace the current ConnectedQ by the faster version, we should see a significant speedup in the new EulerianQ as well. It is quite amazing how slow the old version of EulerianQ is; I suppose I should not use that as a benchmark!

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

```
Timing[ EulerianQ[g]; ]
```

```
{0.734 Second, Null}
```

```
h = DiscreteMath`OldCombinatorica`GridGraph[30, 30];
```

```
Timing[ DiscreteMath`OldCombinatorica`EulerianQ[ h ]; ]
```

```
{9.953 Second, Null}
```

EulerianCycle

```
?EulerianCycle
```

```
EulerianCycle[g] finds an Eulerian circuit of g if one exists.
```

```
EulerianCycle[CompleteGraph[4, 4]]
```

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

```
EulerianCycle[Hypercube[4]]
```

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

```
g = Table[Hypercube[i], {i, 3, 10}];
```

```
Table[Timing[ EulerianCycle[ g[[i]] ];], {i, 5}]
```

```
{{0. Second, Null}, {0.062 Second, Null},
 {0.016 Second, Null}, {1.14 Second, Null}, {0.063 Second, Null}}
```

```
Table[Timing[ ExtractCycles[ g[[i]] ];], {i, 5}]
```

```
{{0.016 Second, Null}, {0.062 Second, Null},
 {0.625 Second, Null}, {1.047 Second, Null}, {18.609 Second, Null}}
```

The seemingly strange non−monotonicbehaviour of the EulerianCycle function is because of the fact that Hypercubes with odd dimension are non−Eulerian,while the rest are. EulerianCycle starts by first checking if the given graph is Eulerian.

The bottleneck for this function also is ExtractCycles, when that is speeded up, EulerianCycle will be faster. Of course, the new version of the function is faster than the old version, but again that is no big consolation.

```
h = Table[DiscreteMath`OldCombinatorica`Hypercube[i], {i, 3, 7}];
```

```
Table[Timing[ DiscreteMath`OldCombinatorica`EulerianCycle[ h[[i]] ];], {i, 5}]
```

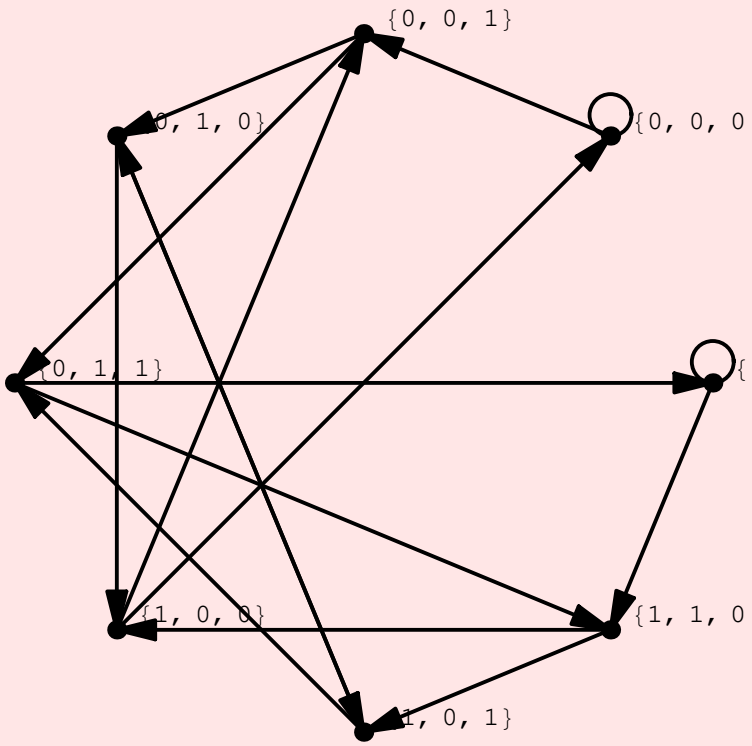
```
{{0.016 Second, Null}, {0.063 Second, Null},
 {0.031 Second, Null}, {1.39 Second, Null}, {0.313 Second, Null}}
```

DeBruijnGraph

```
?DeBruijnGraph
```

```
DeBruijnGraph[m, n] constructs the n-dimensional De Bruijn graph with m
  symbols. DeBruijnGraph[alph, n] constructs the n-dimensional De
  Bruijn graph with symbols from alpha. In the latter form, the
  vertices of the graph are labeled by length (n-1) strings on alph.
```

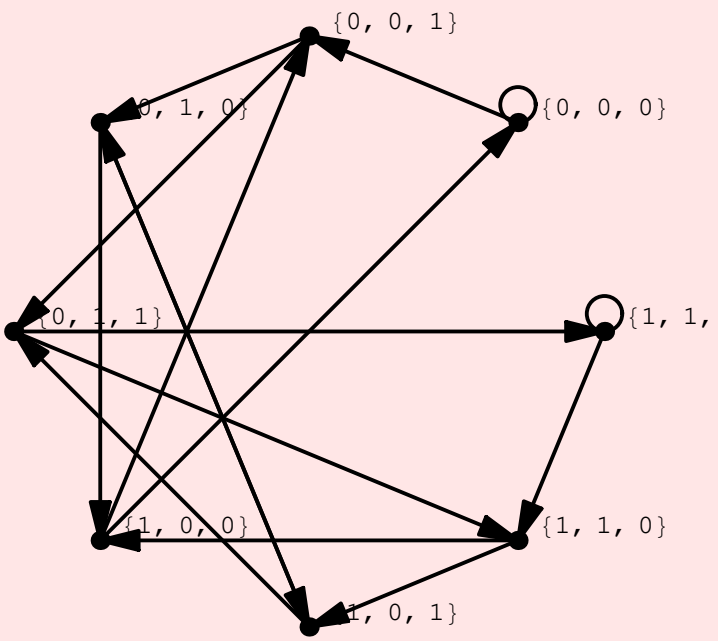
```
ShowGraph[t = DeBruijnGraph[{0, 1}, 4]]
```




```
- Graphics -
```

```
states = Map[FromCharacterCode, Map[(# + 48) &, Strings[{0, 1}, 3]]];
```

```
ShowGraph[SetGraphOptions[t,
  Table[{{i}, VertexLabel -> states[[i]]}, {i, Length[states]}]],
 PlotRange -> {{-0.1, 1.2}, {-0.1, 1.2}}]
```




```
- Graphics -
```

```
EulerianQ[t]
```

```
True
```

```
states[[EulerianCycle[t]]]
```

```
{000, 001, 010, 101, 011, 111, 111,
 110, 101, 010, 100, 001, 011, 110, 100, 000, 000}
```

In my view, it is more important to construct a DeBruijn graph, than a DeBruijn sequence –in fact it is simply one extra line of code to go from a DeBruijn graph to a DeBruijn sequence. So for now, I have replaced the function DeBruijnSequence by the new DeBruijnGraph. Of course we could always have both the graph and the sequence function.

Along with the DeBruijn graph and Hypercube, we might also consider other interconnection networks like the shuffle–exchange, butterfly network etc.

```
Table[Timing[ DeBruijnGraph[ {0, 1}, i];], {i, 3, 10}]
```

```
{{0.015 Second, Null}, {0. Second, Null},
 {0.016 Second, Null}, {0.047 Second, Null}, {0.093 Second, Null},
 {0.266 Second, Null}, {1. Second, Null}, {4.031 Second, Null}}
```

```
Table[Timing[ DeBruijnGraph[ 2, i];], {i, 3, 10}]
```

```
{{0. Second, Null}, {0.016 Second, Null},
 {0. Second, Null}, {0.016 Second, Null}, {0.062 Second, Null},
 {0.219 Second, Null}, {0.828 Second, Null}, {3.328 Second, Null}}
```

We are able to constuct a 10−dimensionalDeBruijn graph (with 1024 vertices) in 4.29 seconds −not bad −is there scope for improvement?

HamiltonianCycle

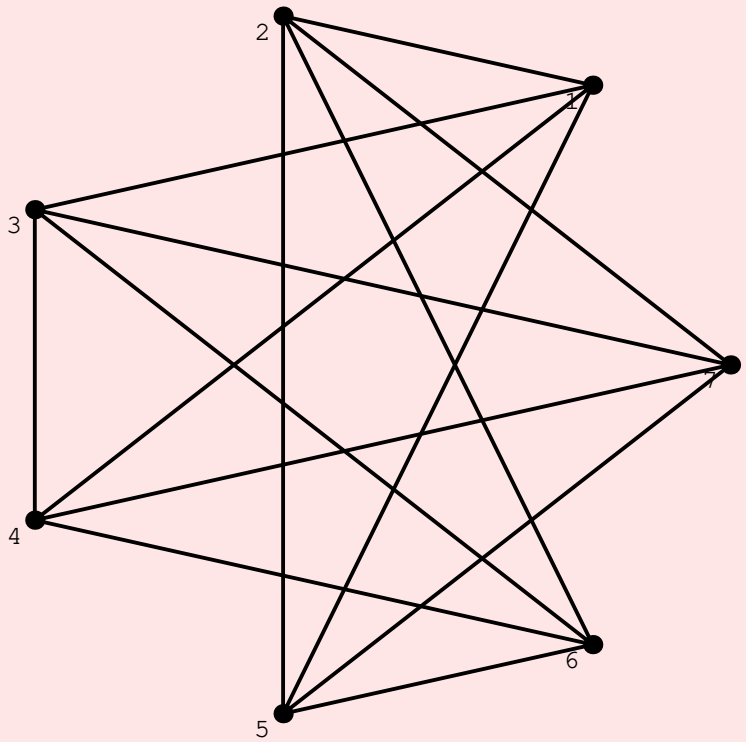
```
?HamiltonianCycle
```

```
HamiltonianCycle[g] finds a Hamiltonian cycle in graph g if one exists.
  HamiltonianCycle[g, All] gives all Hamiltonian cycles of graph g.
```

```
HamiltonianCycle[CompleteGraph[3, 3], All]
```

```
{{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}}
```

```
ShowGraph[t = RegularGraph[4, 7], VertexNumber -> On]
```




```
- Graphics -
```

```
HamiltonianCycle[t]
```

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

```
HamiltonianCycle[AddEdges[GridGraph[5, 5], {{{1, 25}}}]]
```

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

```
HamiltonianCycle[ GridGraph[5, 5] ]
```

```
{}
```

```
g = Table[ GridGraph [i, i], {i, 3, 6}];
```

```
Table [ Timing[ HamiltonianCycle[ g[[i]] ]; ], {i, 4}]
```

```
{{0.016 Second, Null}, {0.016 Second, Null},
 {25.265 Second, Null}, {19.922 Second, Null}}
```

```
h = Table [ DiscreteMath`OldCombinatorica`GridGraph[i, i], {i, 3, 6}];
```

```
Table [
 Timing[ DiscreteMath`OldCombinatorica`HamiltonianCycle[ h[[i]] ]; ], {i, 4}]
```

```
{{0.016 Second, Null}, {0.016 Second, Null},
 {25.359 Second, Null}, {19.812 Second, Null}}
```

There is virtually no difference between the running times of the old version and the new version of Hamiltonian Cycle. Again, I think this is a candidate for improvement.

SetEdgeWeights

```
? 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.
```

This function assigns random weights to edges or vertices of a graph.

```
Edges[SetEdgeWeights[GridGraph[5, 5]], EdgeWeight]
```

```
{{{1, 2}, 0.616363}, {{2, 3}, 0.242657}, {{3, 4}, 0.0951651},
 {{4, 5}, 0.174373}, {{6, 7}, 0.308184}, {{7, 8}, 0.228384}, {{8, 9}, 0.869308},
 {{9, 10}, 0.317166}, {{11, 12}, 0.0589049}, {{12, 13}, 0.947536},
 {{13, 14}, 0.784791}, {{14, 15}, 0.34625}, {{16, 17}, 0.956441},
 {{17, 18}, 0.841366}, {{18, 19}, 0.203512}, {{19, 20}, 0.68216},
 {{21, 22}, 0.516524}, {{22, 23}, 0.0789837}, {{23, 24}, 0.0110615},
 {{24, 25}, 0.108073}, {{1, 6}, 0.0675505}, {{2, 7}, 0.49902},
 {{3, 8}, 0.0746715}, {{4, 9}, 0.393005}, {{5, 10}, 0.451187},
 {{6, 11}, 0.256363}, {{7, 12}, 0.979506}, {{8, 13}, 0.218632},
 {{9, 14}, 0.143003}, {{10, 15}, 0.0279786}, {{11, 16}, 0.110199},
 {{12, 17}, 0.901465}, {{13, 18}, 0.084098}, {{14, 19}, 0.0804427},
 {{15, 20}, 0.325408}, {{16, 21}, 0.555215}, {{17, 22}, 0.127657},
 {{18, 23}, 0.239077}, {{19, 24}, 0.121896}, {{20, 25}, 0.873055}}
```

```
Edges[SetEdgeWeights[GridGraph[5, 5],
  WeightingFunction -> RandomInteger, WeightRange -> {10, 20}], EdgeWeight]
```

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

```
Edges[SetEdgeWeights[GridGraph[5, 5],
  WeightingFunction -> Random, WeightRange -> {-10, 10}], EdgeWeight]
```

```
{{{1, 2}, -6.03332}, {{2, 3}, -1.85982}, {{3, 4}, -2.73121}, {{4, 5}, 3.06176},
 {{6, 7}, -6.96123}, {{7, 8}, -8.14157}, {{8, 9}, 4.77624}, {{9, 10}, -1.01041},
 {{11, 12}, 6.59344}, {{12, 13}, -9.27523}, {{13, 14}, -0.551626},
 {{14, 15}, -5.58674}, {{16, 17}, -5.45018}, {{17, 18}, -1.82038},
 {{18, 19}, -5.43148}, {{19, 20}, 5.90203}, {{21, 22}, -9.33851},
 {{22, 23}, -5.2686}, {{23, 24}, 0.855272}, {{24, 25}, -5.7657},
 {{1, 6}, -7.4562}, {{2, 7}, -9.77859}, {{3, 8}, -5.48283}, {{4, 9}, -9.96557},
 {{5, 10}, 8.57712}, {{6, 11}, 2.08123}, {{7, 12}, 7.24838}, {{8, 13}, -3.02733},
 {{9, 14}, 5.53835}, {{10, 15}, 0.222798}, {{11, 16}, -7.52786},
 {{12, 17}, 7.98309}, {{13, 18}, 8.94491}, {{14, 19}, -0.501968},
 {{15, 20}, 3.02376}, {{16, 21}, 3.56983}, {{17, 22}, 4.39509},
 {{18, 23}, -8.68159}, {{19, 24}, -1.54476}, {{20, 25}, 7.6678}}
```

```
Edges[SetEdgeWeights[GridGraph[3, 3],
  WeightingFunction -> Euclidean], EdgeWeight]
```

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

```
(*GetEdgeWeights[SetEdgeWeights[GridGraph[3, 3], WeightingFunction->L[1]]]*)
```

```
{L[1][{1., 1.}, {2., 1.}], L[1][{2., 1.}, {3., 1.}], L[1][{1., 2.}, {2., 2.}],
 L[1][{2., 2.}, {3., 2.}], L[1][{1., 3.}, {2., 3.}], L[1][{2., 3.}, {3., 3.}],
 L[1][{1., 1.}, {1., 2.}], L[1][{2., 1.}, {2., 2.}], L[1][{3., 1.}, {3., 2.}],
 L[1][{1., 2.}, {1., 3.}], L[1][{2., 2.}, {2., 3.}], L[1][{3., 2.}, {3., 3.}]}
```

```
GetEdgeWeights[
 SetEdgeWeights[RandomGraph[10, .3], WeightingFunction -> Euclidean]]
```

```
{1.90211, 1.17557, 1.61803, 0.618034, 0.618034, 1.17557,
 1.90211, 1.17557, 0.618034, 1.61803, 1.90211, 1.61803, 1.17557}
```

SetVertexWeights

```
? SetVertexWeights
```

```
SetVertexWeights[g] assigns random real non-negative weights to vertices
  in g. SetVertexWeights[g, Random, type, range] assigns random
  weights of specified type in the specified range to vertices
  in g. SetVertexWeights[g, w] assigns the weights in weight
  list w to the vertices in g. SetVertexWeights[g, v, w] assigns
  the weights in weight list w to the vertices in vertex list v.
```

*BUG: SetVertexWeights should have the same syntax as setEdgeWeights.

```
GetVertexWeights[
 SetVertexWeights[GridGraph[3, 3], Random, Integer, {10, 20}]]
```

```
{20, 16, 14, 11, 19, 10, 11, 13, 11}
```

```
GetVertexWeights[SetVertexWeights[GridGraph[3, 3]]]
```

```
{0.611132, 0.160093, 0.110834, 0.764982,
 0.543582, 0.661074, 0.0361625, 0.371977, 0.0923942}
```

GetEdgeWeights

```
? GetEdgeWeights
```

```
GetEdgeWeights[g] returns the list of weights of the edges of g.
```

```
g = SetEdgeWeights[GridGraph[5, 5],
   WeightingFunction -> RandomInteger, WeightRange -> {0, 10}];
```

```
GetEdgeWeights[g]
```

```
{2, 10, 6, 8, 5, 6, 2, 3, 6, 2, 6, 10, 1, 2, 8, 6, 5, 4, 10,
 2, 5, 10, 4, 3, 10, 10, 5, 6, 1, 3, 1, 6, 0, 4, 2, 7, 8, 7, 4, 3}
```

GetVertexWeights

```
?GetVertexWeights
```

```
GetVertexWeights[g] returns the list of weights of vertices of g.
```

```
GetVertexWeights[g]
```

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

```
g = SetVertexWeights[g, Random, Integer, {100, 200}];
```

```
GetVertexWeights[g]
```

```
{178, 150, 186, 129, 156, 103, 107, 187, 118, 119, 131, 150,
 149, 178, 158, 149, 189, 119, 130, 197, 157, 162, 121, 199, 130}
```

CostOfPath

```
?CostOfPath
```

```
CostOfPath[g, p] sums up the weights
  of the edges in graph g defined by the path p.
```

```
g = SetEdgeWeights[GridGraph[4, 4],
  WeightingFunction -> RandomInteger, WeightRange -> {1, 5}]
```

```
-Graph:<24, 16, Undirected>-
```

```
CostOfPath[g, {1, 2, 3, 4, 8, 12}]
```

```
17
```

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

```
∞
```

TravelingSalesman

```
?TravelingSalesman
```

```
TravelingSalesman[g] finds the
  optimal traveling salesman tour in graph g.
```

```
g = SetEdgeWeights[Cycle[10],
  WeightingFunction -> RandomInteger, WeightRange -> {1, 3}]
```

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

```
TravelingSalesman[g]
```

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

```
GetEdgeWeights[g = SetEdgeWeights[CompleteGraph[4],
   WeightingFunction -> RandomInteger, WeightRange -> {1, 100}]]
```

```
{39, 97, 13, 22, 71, 70}
```

```
TravelingSalesman[g]
```

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

```
GetEdgeWeights[g = SetEdgeWeights[Wheel[6], WeightingFunction -> Euclidean]]
```

```
{1., 1., 1., 1., 1., 1.17557, 1.17557, 1.17557, 1.17557, 1.17557}
```

```
p = TravelingSalesman[g]
```

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

```
CostOfPath[g, p]
```

```
6.70228
```

```
CostOfPath[g, {1, 2, 4, 3, 5, 6, 1}]
```

```
∞
```

```
CostOfPath[g, {1, 2, 6, 3, 4, 5, 1}]
```

```
6.70228
```

```
g = Table[SetEdgeWeights[CompleteGraph[i],
    WeightingFunction -> RandomInteger, WeightRange -> {1, 100}], {i, 5, 8}];
```

```
Table[ Timing[ TravelingSalesman[ g[[i]] ]; ], {i, 4}]
```

```
{{0.078 Second, Null}, {0.829 Second, Null},
 {6.171 Second, Null}, {32.36 Second, Null}}
```

HamiltonianCycle and TravelingSalesman are very similar functions and any ideas for speeding up one can be applied to the other. For TravelingSalesman, there are different heuristics that can be used.

TriangleInequalityQ

```
?TriangleInequalityQ
```

```
TriangleInequalityQ[g] yields True if the weights assigned
  to the edges of graph g satisfy the triangle inequality.
```

I don't see this function used very much and so this is another candidate for deletion. I also wonder if I can implement this more efficiently –I simply construct a matrix of edge weights and then use the old code.

```
GetEdgeWeights[g = SetEdgeWeights[GridGraph[2, 2]]]
```

```
{0.40471, 0.0566561, 0.153345, 0.949391}
```

```
TriangleInequalityQ[g]
```

```
True
```

The above function returns true because there are no triangle in the grid graph!

```
GetEdgeWeights[g = SetEdgeWeights[CompleteGraph[5]]]
```

```
{0.376732, 0.946457, 0.25188, 0.865293, 0.296289,
 0.62105, 0.696664, 0.737637, 0.0572121, 0.499154}
```

```
TriangleInequalityQ[g]
```

```
False
```

As expected, randomly chosen edge weights will typically not satisfy the triangle inequality.

Below I set weights in K_4 to satisfy the triangle inequality.

```
g = SetGraphOptions[CompleteGraph[4],
  {{{{1, 2}, {2, 3}, {3, 4}, {1, 4}}, EdgeWeight -> 1},
   {{{2, 4}, {1, 3}}, EdgeWeight -> 1.5}}]
```

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

```
TriangleInequalityQ[g]
```

```
True
```

```
g = SetGraphOptions[g, EdgeDirection -> On]
```

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

```
TriangleInequalityQ[g]
```

```
True
```

TravelingSalesmanBounds

```
?TravelingSalesmanBounds
```

```
TravelingSalesmanBounds[g] gives upper and lower bounds
  on the minimum cost traveling salesman tour of graph g.
```

```
TravelingSalesmanBounds[Wheel[5]]
```

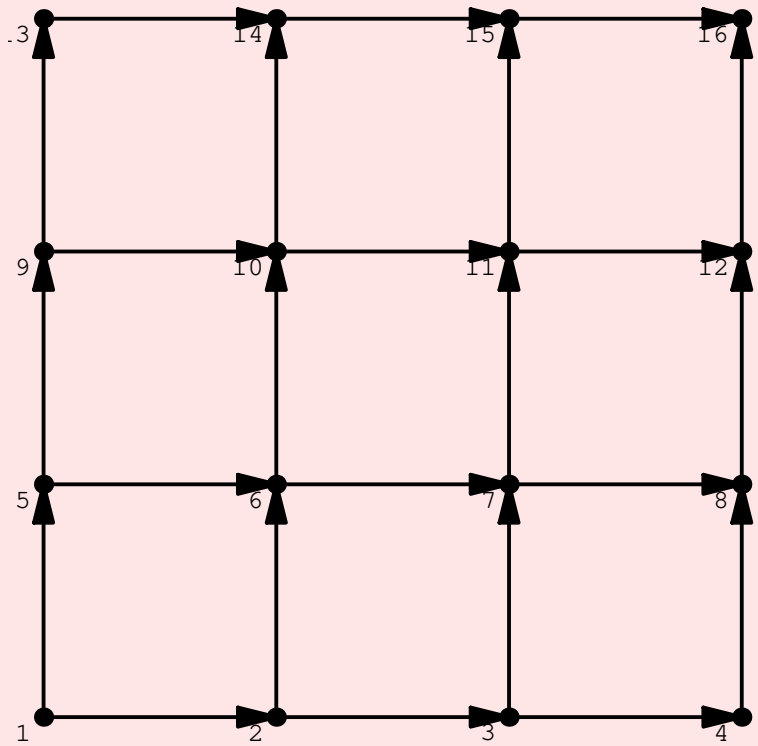
```
{17, 5}
```

TransitiveClosure

```
?TransitiveClosure
```

```
TransitiveClosure[g] finds the transitive
  closure of graph g, the supergraph of g that contains
  edge {x, y} if and only if there is a path from x to y.
```

```
ShowGraph[g = SetGraphOptions[GridGraph[4, 4], EdgeDirection -> On],
 VertexNumber -> On]
```



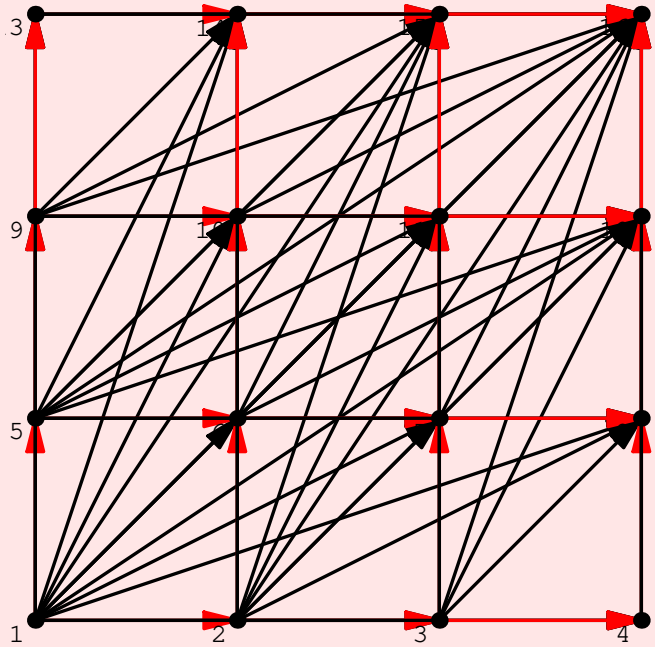

```
- Graphics -
```

Here is an attempt to make the edges in the orginal graph show up in a different color as compared to the new edges added during by transitive closure. It does not quite work as one would want because of "hidden" edges such as {1, 3}, {1, 4}, {1, 9}, amd {1, 13} that are added by transitive closure AND because of the order in which the edges are rendered.

```
Edges[g]
```

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

```
ShowGraph[gg = TransitiveClosure[g],
 Append[Edges[g], EdgeColor -> Red], VertexNumber → On]
```




```
- Graphics -
```

```
Edges[gg]
```

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

```
g = DiscreteMath`OldCombinatorica`GridGraph[10, 10];
Timing[ DiscreteMath`OldCombinatorica`TransitiveClosure[g];]
```

```
{13.969 Second, Null}
```

```
g = GridGraph[10, 10];
```

```
Timing[ TransitiveClosure[g]; ]
```

```
{13.953 Second, Null}
```

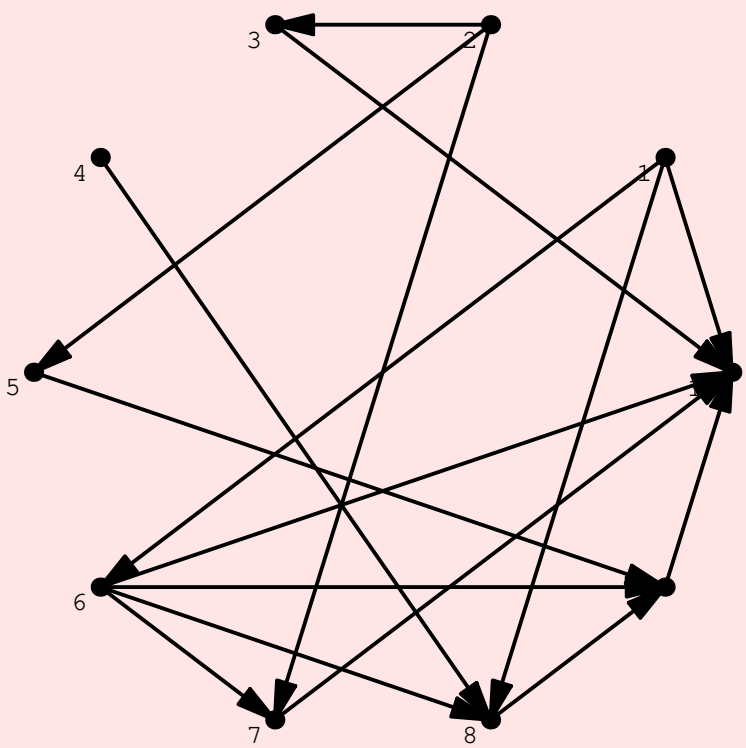
As expected, the new TransitiveClosure is slower than the old TransitiveClosure because the new version converts edge lists into an adjacency matrix representation and then simply uses the old code. I might be able to do this faster by not going into adjacency matrix representation??

TransitveQ

```
?TransitiveQ
```

```
TransitiveQ[g] yields True if graph g defines a transitive relation.
```

```
ShowGraph[g = SetGraphOptions[RandomGraph[10, 0.3], EdgeDirection -> On],
 VertexNumber -> On]
```




```
- Graphics -
```

```
TransitiveQ[g]
```

```
False
```

```
TransitiveQ[TransitiveClosure[g]]
```

```
True
```

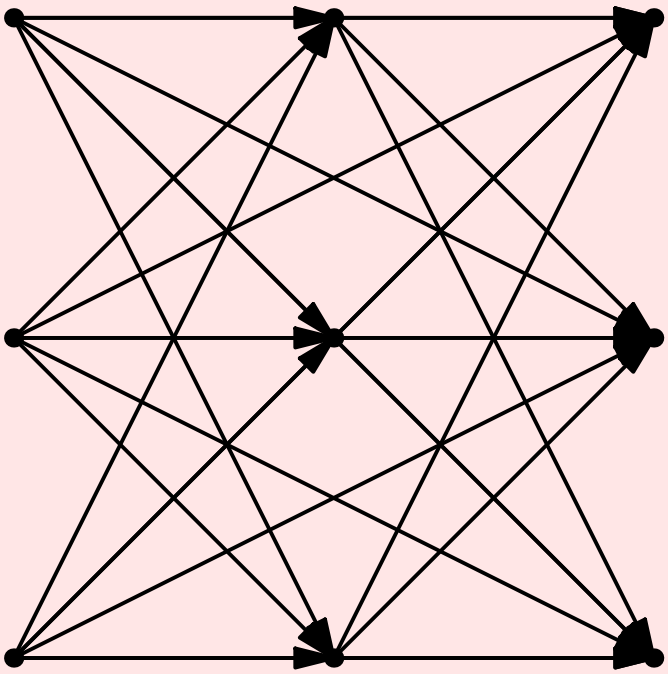
```
TransitiveQ[SetGraphOptions[CompleteGraph[10], EdgeDirection -> On]]
```

```
True
```

```
TransitiveQ[g = SetGraphOptions[CompleteGraph[3, 3, 3], EdgeDirection -> On]]
```

```
True
```

```
ShowGraph[g]
```



```
- Graphics -
```

ReflexiveQ

```
?ReflexiveQ
```

```
ReflexiveQ[g] yields True if the adjacency
  matrix of g represents a reflexive binary relation.
```

```
ReflexiveQ[g = CompleteGraph[10]]
```

```
False
```

```
g = AddEdges[g, Table[{{i, i}}, {i, 10}]];
```

```
ReflexiveQ[g]
```

```
True
```

```
g = DeleteEdges[g, {{1, 1}}];
```

```
ReflexiveQ[g]
```

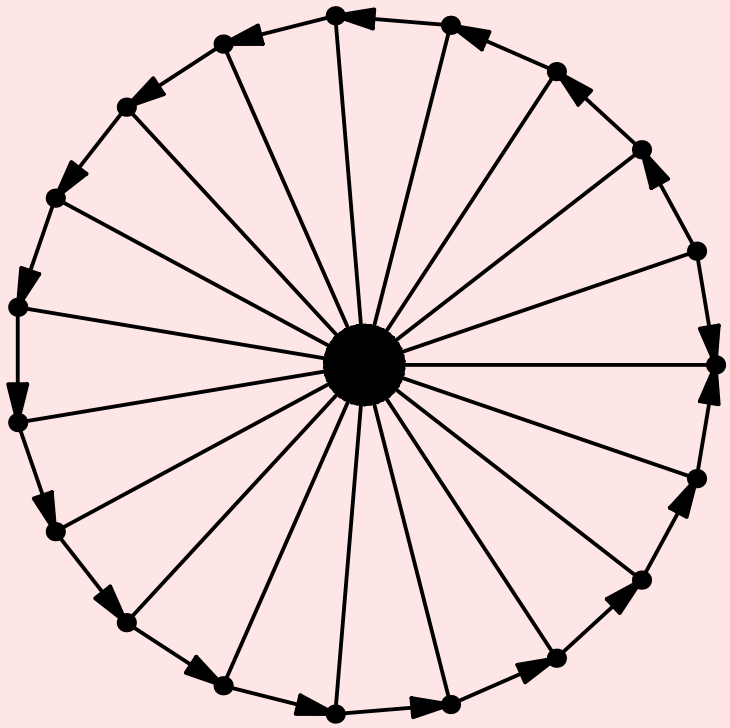
```
False
```

AntiSymmetricQ

```
?AntiSymmetricQ
```

```
AntiSymmetricQ[g] yields True if the adjacency
  matrix of g represents an anti-symmetric binary relation.
```

```
ShowGraph[g = SetGraphOptions[Wheel[20], EdgeDirection -> On]]
```

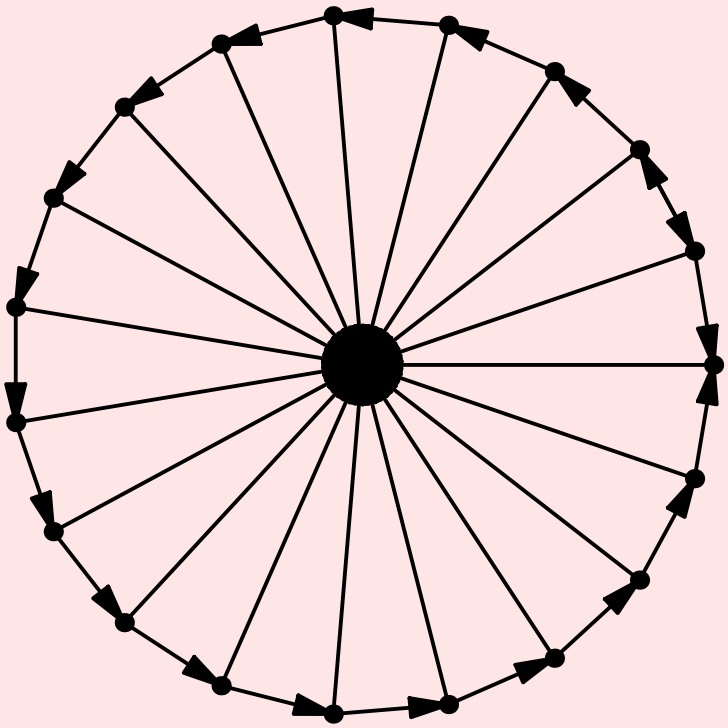


```
- Graphics -
```

```
AntiSymmetricQ[g]
```

```
True
```

```
ShowGraph[g = AddEdges[g, {{{2, 1}}}]]
```



```
- Graphics -
```

```
AntiSymmetricQ[g]
```

```
False
```

PartialOrderQ

```
?PartialOrderQ
```

```
PartialOrderQ[g] yields True if the binary relation defined
  by the adjacency matrix of graph g is a partial order,
  meaning it is transitive, reflexive, and anti-symmetric.
```

```
PartialOrderQ[g = GridGraph[3, 3]]
```

```
False
```

```
g  = TransitiveClosure[g];
```

```
Edges[g]
```

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

```
PartialOrderQ[g]
```

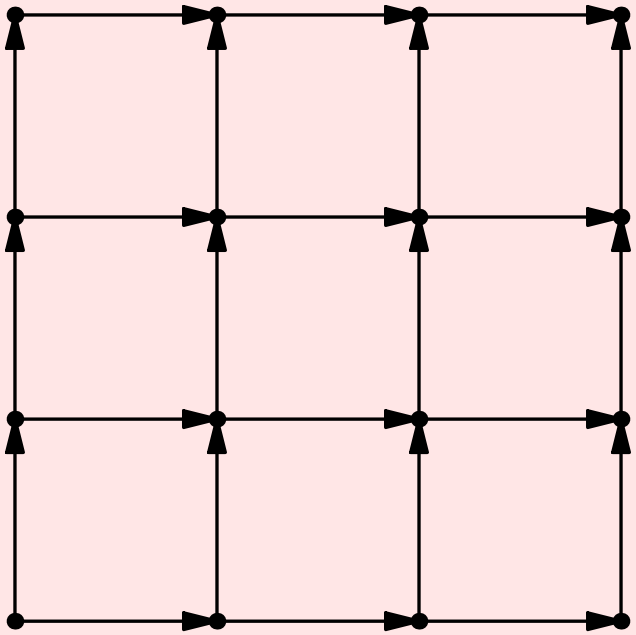
```
True
```

TransitiveReduction

```
?TransitiveReduction
```

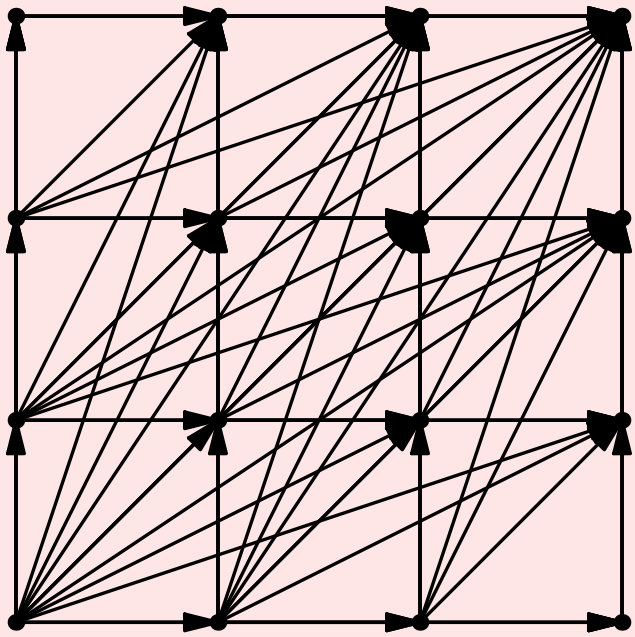
```
TransitiveReduction[g] finds the smallest
  graph which has the same transitive closure as g.
```

```
ShowGraph[TransitiveReduction[
  g = SetGraphOptions[GridGraph[4, 4], EdgeDirection -> On]]]
```



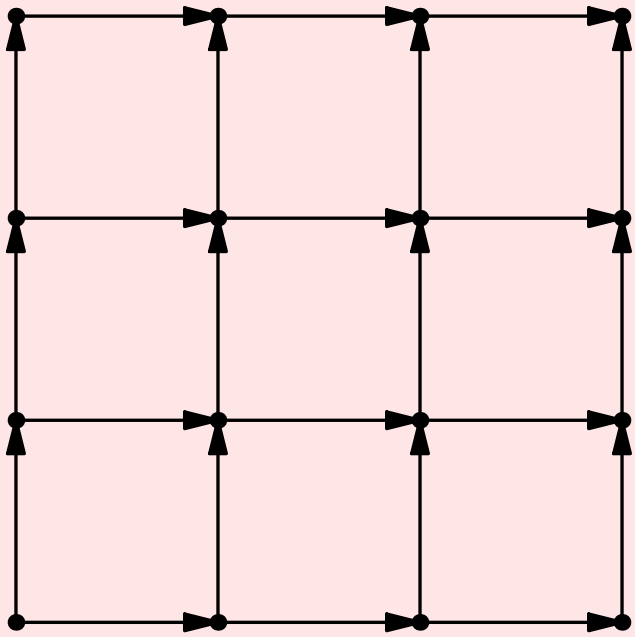
```
- Graphics -
```

```
ShowGraph[g = TransitiveClosure[g]]
```



```
- Graphics -
```

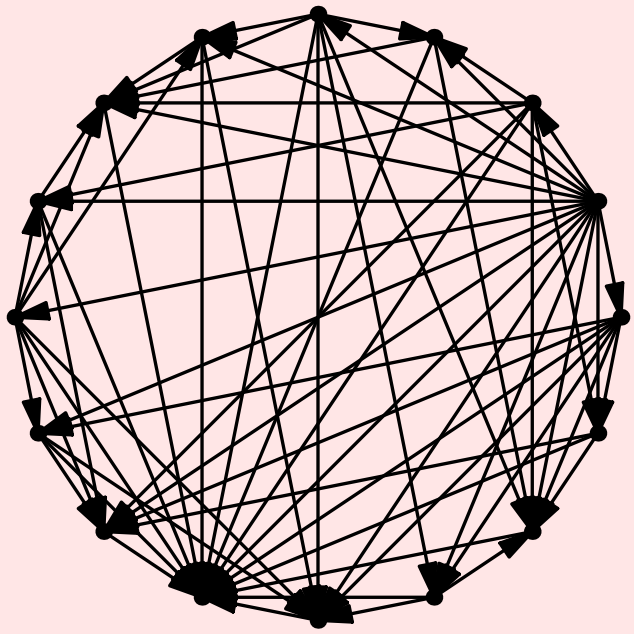
```
ShowGraph[TransitiveReduction[g]]
```



```
- Graphics -
```

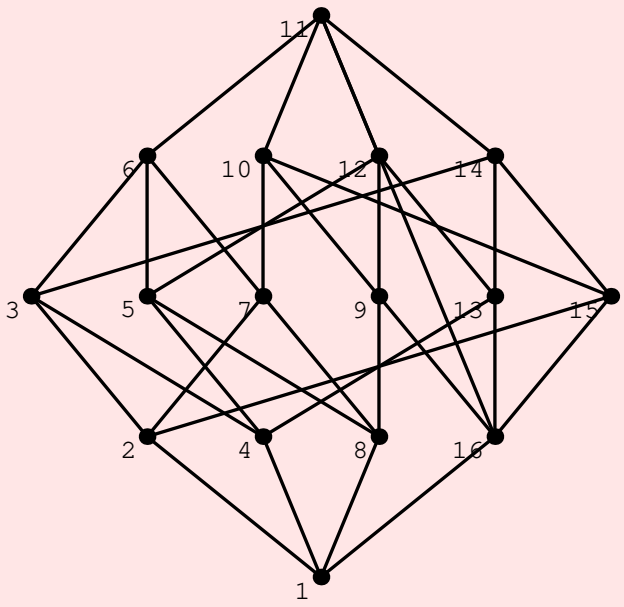
HasseDiagram

```
ShowGraph[
 s = MakeGraph[Subsets[4], ((Intersection[#2, #1] === #1) && (#1 != #2)) &]]
```



```
- Graphics -
```

```
ShowGraph[t = HasseDiagram[s], VertexNumber -> On]
```



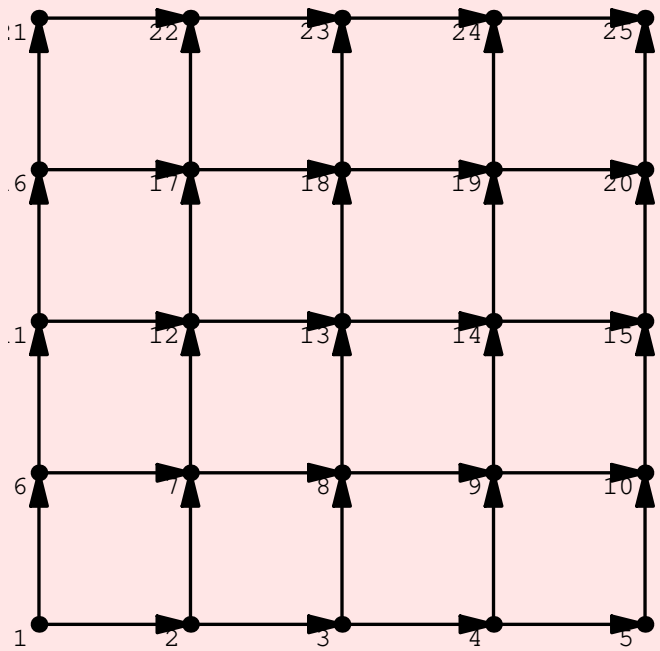

```
- Graphics -
```

TopologicalSort

```
? TopologicalSort
```

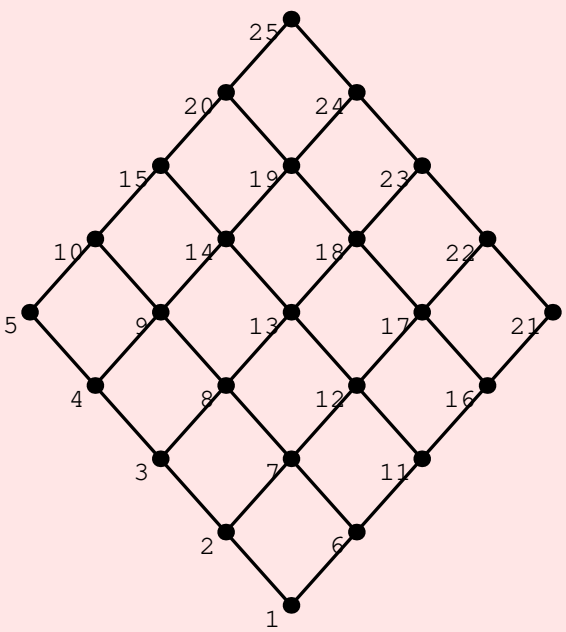
```
TopologicalSort[g] gives a permutation of the
  vertices of directed acyclic graph g such that an edge (
  i, j) implies that vertex i appears before vertex j.
```

```
ShowGraph[g = SetGraphOptions[GridGraph[5, 5], EdgeDirection -> On],
 VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[HasseDiagram[g], VertexNumber -> On]
```

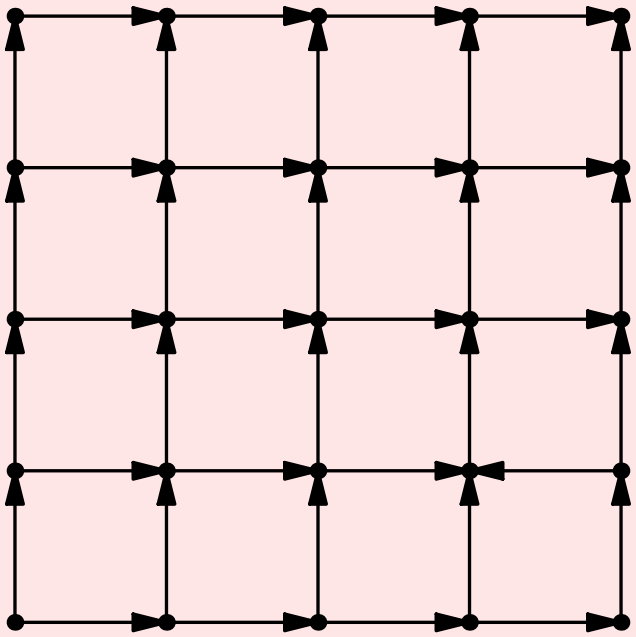



```
- Graphics -
```

```
TopologicalSort[g]
```

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

```
ShowGraph[g = AddEdges[DeleteEdges[g, {{9, 10}}], {{{10, 9}}}]]
```

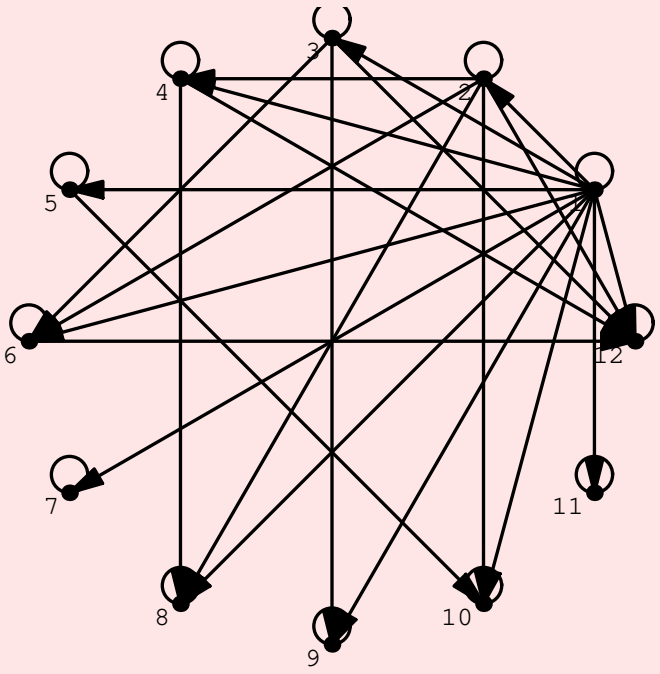


```
- Graphics -
```

```
TopologicalSort[g]
```

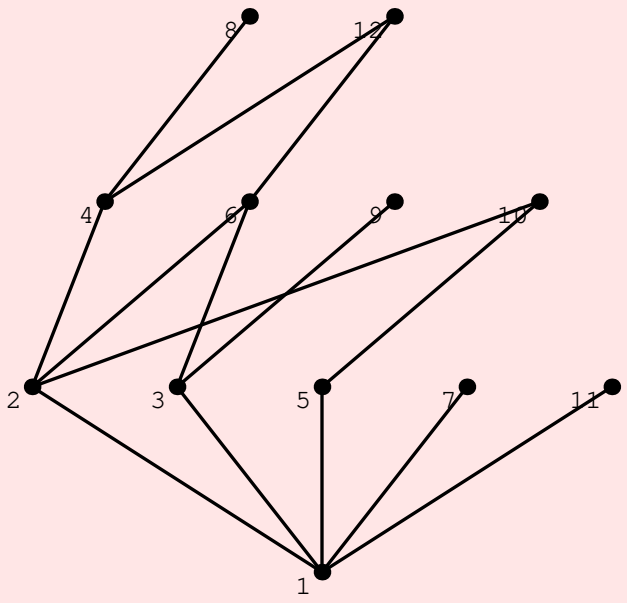
```
{1, 2, 6, 3, 7, 11, 4, 8, 12, 16, 5, 13,
 17, 21, 10, 18, 22, 9, 23, 14, 15, 19, 20, 24, 25}
```

```
ShowGraph[g = MakeGraph[Range[12], (Mod[#2, #1] == 0) &], VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[HasseDiagram[g], VertexNumber -> On]
```




```
- Graphics -
```

```
TopologicalSort[g]
```

```
{1, 2, 3, 5, 7, 11, 4, 6, 9, 10, 8, 12}
```

ChromaticPolynomial

```
?ChromaticPolynomial
```

```
ChromaticPolynomial[g, z] gives the chromatic polynomial P(z) of graph
  g, which counts the number of ways to color g with, at most, z colors.
```

```
g = CompleteGraph[5];
```

```
ChromaticPolynomial[g, z]
```

```
(-4 + z) (-3 + z) (-2 + z) (-1 + z) z
```

```
g = GraphUnion[10, CompleteGraph[1]];
```

```
ChromaticPolynomial[g, z]
```

$$z^{10}$$

```
ChromaticPolynomial[RandomTree[10], z]
```

$$(-1+z)^9\, z$$

```
g = DeleteEdges[CompleteGraph[5], {{1, 2}}];
```

```
ChromaticPolynomial[g, z]
```

$$(-3+z)\,(-2+z)\,(-1+z)\,z+(-4+z)\,(-3+z)\,(-2+z)\,(-1+z)\,z$$

```
ChromaticPolynomial[Cycle[5], z]
```

$$4\,z-10\,z^2+10\,z^3-5\,z^4+z^5$$

```
Table[{Timing[ DiscreteMath`OldCombinatorica`ChromaticPolynomial[
     DiscreteMath`OldCombinatorica`Cycle[i], z];],
  Timing[ChromaticPolynomial[Cycle[i], z];]}, {i, 5, 10}]
```

```
{{{0.047 Second, Null}, {0.015 Second, Null}},
 {{0.063 Second, Null}, {0.047 Second, Null}},
 {{0.109 Second, Null}, {0.109 Second, Null}},
 {{0.235 Second, Null}, {0.203 Second, Null}},
 {{0.484 Second, Null}, {0.422 Second, Null}},
 {{1. Second, Null}, {0.844 Second, Null}}}
```

The speedup is not exciting! Clearly,more work is needed here.

```
Expand[wheelpoly[z_] = ChromaticPolynomial[Wheel[5], z]]
```

$$14\,z-31\,z^2+24\,z^3-8\,z^4+z^5$$

```
Table[wheelpoly[z], {z, 1, 7}]
```

```
{0, 0, 6, 72, 420, 1560, 4410}
```

ChromaticNumber

```
?ChromaticNumber
```

```
ChromaticNumber[g] gives the chromatic number of the graph, which
  is the fewest number of colors necessary to color the graph.
```

```
ChromaticNumber[Cycle[7]]
```

```
3
```

```
ChromaticNumber[CompleteGraph[4, 5]]
```

```
2
```

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

```
Timing[ ChromaticNumber[g]; ]
```

```
$Aborted
```

This function is hopelessly slow since it uses ChromaticPolynomials. We have to find other ways of finding an optimal coloring. I ran out of patience in trying the color the above 5X5 grid graph.

TwoColoring

```
?TwoColoring
```

```
TwoColoring[g] finds a two-coloring of graph g if g is bipartite. It
  returns as assignment of the labels 1 and 2 to vertices. This
  assignment is a valid coloring if and only the graph is bipartite.
```

```
TwoColoring[t = RandomTree[20]]
```

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

```
$RecursionLimit = 100000;
```

```
TwoColoring[CompleteGraph[3, 3, 3]]
```

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

```
g = Table[DiscreteMath`OldCombinatorica`GridGraph[i, i], {i, 10, 30, 10}];
```

```
h = Table[GridGraph[i, i], {i, 10, 30, 10}];
```

```
Table[{Timing[DiscreteMath`OldCombinatorica`TwoColoring[g[[i]]];],
  Timing[TwoColoring[h[[i]]];]}, {i, 3}]
```

```
{{{0.062 Second, Null}, {0.031 Second, Null}},
 {{0.735 Second, Null}, {0.187 Second, Null}},
 {{3.547 Second, Null}, {0.594 Second, Null}}}
```

The new version of BreadthFirstTraversal returns levels of vertices as well. This can be used to obtain a one−lineTwoColoring function. However, as can be seen from below, the new version is slightly slower, even though both versions are much faster than the old version of two coloring.

```
MyTC[ g_Graph] := Map[If[EvenQ[#], 1, 2] &, BreadthFirstTraversal[g, 1, Level]
```

```
MyTC[ t ]
```

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

```
Table[ Timing[ MyTC[h[[i]]]; ], {i, 3}]
```

```
{{0.016 Second, Null}, {0.094 Second, Null}, {0.25 Second, Null}}
```

BipartiteQ

```
?BipartiteQ
```

```
BipartiteQ[g] yields True if graph g is bipartite.
```

```
BipartiteQ[Wheel[10]]
```

```
False
```

```
BipartiteQ[Star[10]]
```

```
True
```

```
BipartiteQ[GridGraph[5, 5]]
```

```
True
```

```
BipartiteQ[Cycle[5]]
```

```
False
```

VertexColoring

```
?VertexColoring
```

```
VertexColoring[g] uses Brelaz's heuristic to find a good, but not
  necessarily minimal, vertex coloring of graph g. An option
  Algorithm that can take on the values Brelaz or Optimum is
  allowed. The setting Algorithm -> Brelaz is default, while
  the setting Algorithm -> Optimum forces the algorithm to
  do an exhaustive search and find an optimum vertex coloring.
```

```
VertexColoring[CompleteGraph[4]]
```

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

```
VertexColoring[GridGraph[10, 10]]
```

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

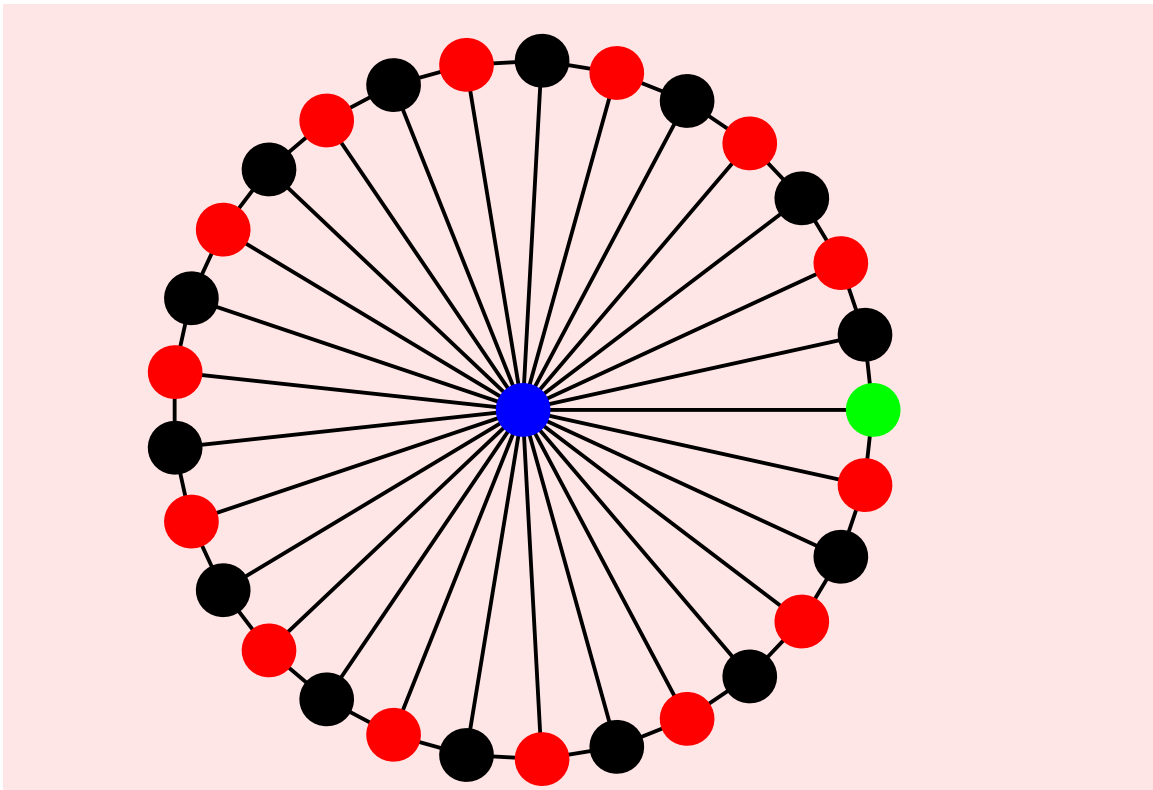
```
c = VertexColoring[ g = Wheel[30] ]
```

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

```
p = Table[Flatten[Position[c, i], 1], {i, Max[c]}]
```

```
{{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27},
 {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28}, {30}, {29}}
```

```
ShowGraph[ Highlight[g, p] ]
```



```
- Graphics -
```

```
g = Table[DiscreteMath`OldCombinatorica`GridGraph[i, i], {i, 5, 30, 10}];
```

```
h = Table[GridGraph[i, i], {i, 5, 30, 10}];
```

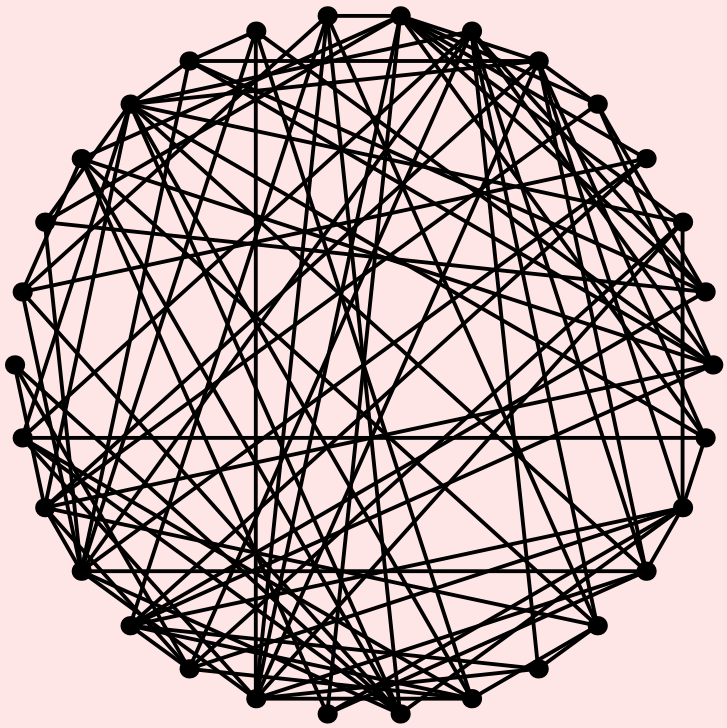
```
Table[{Timing[DiscreteMath`OldCombinatorica`VertexColoring[g[[i]]];],
  Timing[VertexColoring[h[[i]]];]}, {i, 3}]
```

```
$Aborted
```

Originally, I used Steve's code for this and was amazed at how slow both the new and the old implementations were (though the new implementation was just a little faster than the old implementation). I rewrote this function and the speedup is now significant significant!

```
g = ExactRandomGraph[30, 100];
```

```
ShowGraph[g]
```



```
- Graphics -
```

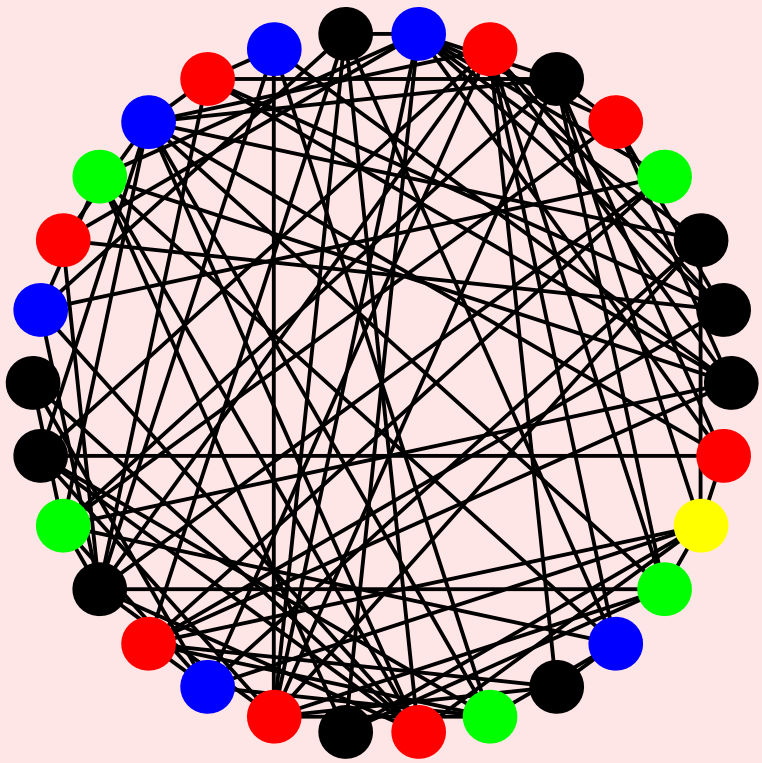
```
c = VertexColoring[g]
```

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

```
p = Table[Flatten[Position[c, i], 1], {i, Max[c]}]
```

```
{{1, 2, 5, 8, 15, 16, 18, 22, 25, 30}, {4, 6, 10, 13, 19, 21, 23, 29},
 {7, 9, 11, 14, 20, 26}, {3, 12, 17, 24, 27}, {28}}
```

```
ShowGraph[Highlight[g, p]]
```



```
- Graphics -
```

EdgeColoring

```
?EdgeColoring
```

```
EdgeColoring[g] uses Brelaz's heuristic to find a
  good, but not necessarily minimal, edge coloring of graph g.
```

```
EdgeColoring[Wheel[10]]
```

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

```
c = EdgeColoring[GridGraph[5, 5]]
```

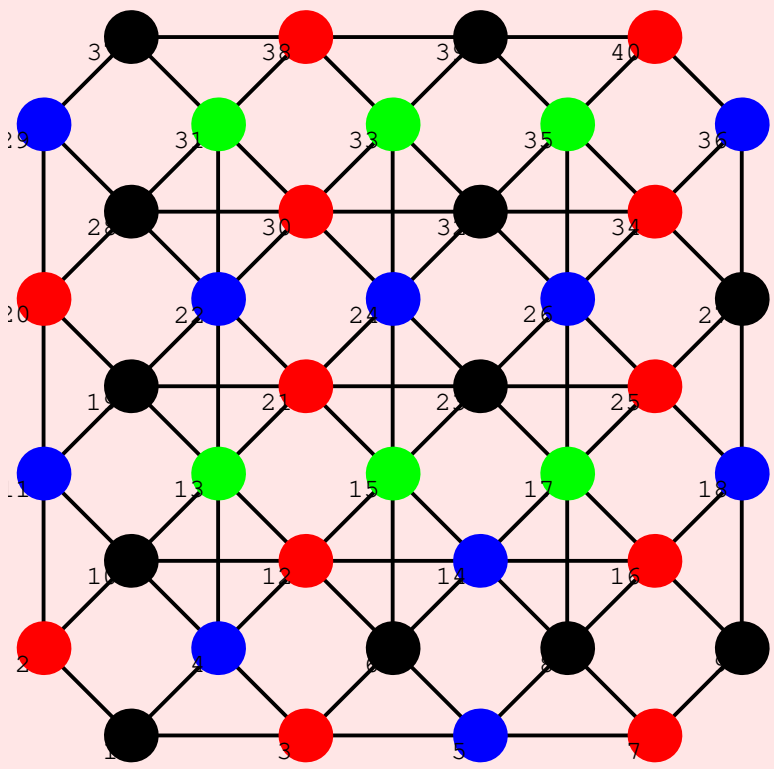
```
{1, 2, 2, 3, 3, 1, 2, 1, 1, 1, 3, 2, 4, 3, 4, 2, 4, 3,
 1, 2, 2, 3, 1, 3, 2, 3, 1, 1, 3, 2, 4, 1, 4, 2, 4, 3, 1, 2, 1, 2}
```

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

```
p = Table[Flatten[Position[c, i], 1], {i, 4}]
```

```
{{1, 6, 8, 9, 10, 19, 23, 27, 28, 32, 37, 39},
 {2, 3, 7, 12, 16, 20, 21, 25, 30, 34, 38, 40},
 {4, 5, 11, 14, 18, 22, 24, 26, 29, 36}, {13, 15, 17, 31, 33, 35}}
```

```
ShowGraph[Highlight[g, p], VertexNumber -> On]
```

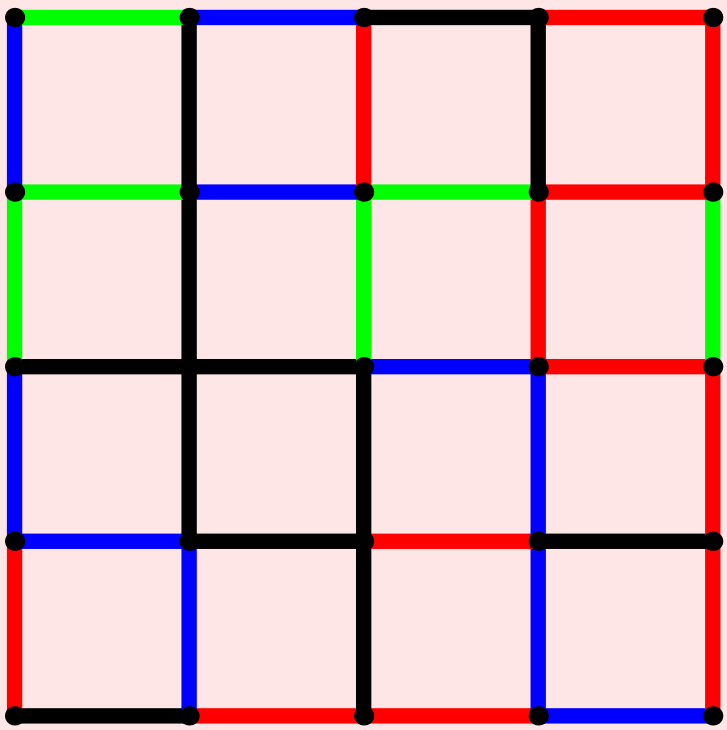


```
- Graphics -
```

```
ep = Map[Edges[g = GridGraph[5, 5]][[#]] &, p]
```

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

```
ShowGraph[Highlight[g, ep]]
```



```
- Graphics -
```

```
g = Table [ GridGraph[i, i], {i, 10, 30, 10}];
```

```
Table[Timing[ EdgeColoring[ g[[i]] ];], {i, 2}]
```

```
{{0.078 Second, Null}, {0.515 Second, Null}}
```

```
Table[ Timing[ LineGraph[ g[[i]] ]; ], {i, 2}]
```

```
{{0.016 Second, Null}, {0.109 Second, Null}}
```

```
h = Table [ DiscreteMath`OldCombinatorica`GridGraph[i, i], {i, 10, 30, 10}];
```

```
Table[Timing[ DiscreteMath`OldCombinatorica`EdgeColoring[ h[[i]] ];], {i, 2}]
```

```
{{2.078 Second, Null}, {123.688 Second, Null}}
```

```
Table[Timing[ DiscreteMath`OldCombinatorica`LineGraph[ h[[i]] ];], {i, 2}]
```

```
{{0.172 Second, Null}, {5.453 Second, Null}}
```

Brelaz's Heuristic has been speeded up and that shows up in the speedup of the new implementation of edge coloring. However, the construction of a line graph is quite slow and when that is speeded up, it will take much less than 22 seconds to color the edges in a 20X20 grid graph. Note that about 18 of the 22 seconds is spent in construction of the line graph. Of course, the new version still beats the old version by far. Notice how in the old version 25 seconds is spent on constructing the line graph, but about 165 seconds is spent in coloring the graph.

EdgeChromaticNumber

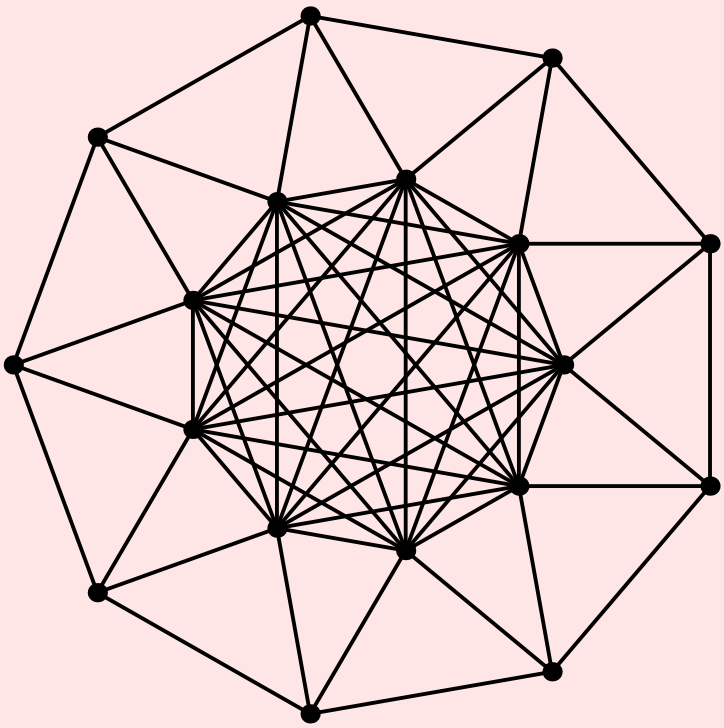
```
?EdgeChromaticNumber
```

```
EdgeChromaticNumber[g] gives the fewest number of
  colors necessary to color each edge of graph g, so that no
  two edges incident on the same vertex have the same color.
```

```
EdgeChromaticNumber[Star[10]]
```

```
9
```

```
ShowGraph[t = LineGraph[Wheel[10]]]
```



```
- Graphics -
```

```
ChromaticPolynomial[t, z]
```

```
$Aborted
```

It takes too long to compute the chromatic polynomial of the above line graph!

MaximumClique

```
?MaximumClique
```

```
MaximumClique[g] finds the largest clique
  in graph g. MaximumClique[g, k] returns a k-clique,
  if such a thing exists in g, otherwise it returns {}.
```

```
MaximumClique[t]
```

```
{3, 5, 7, 9, 11, 13, 15, 17, 18}
```

```
MaximumClique[Turan[6, 4]]
```

```
{1, 3, 5}
```

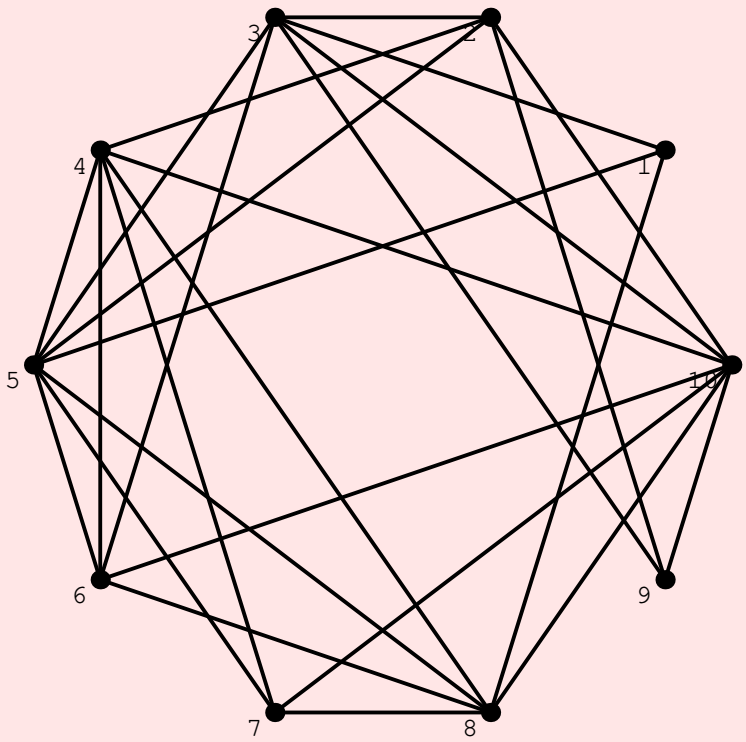
```
MaximumClique[CompleteGraph[10]]
```

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

```
MaximumClique[t = RandomGraph[10, .5]]
```

```
{2, 3, 9, 10}
```

```
ShowGraph[t, VertexNumber -> On]
```




```
- Graphics -
```

```
t = RandomGraph[20, .5];
```

```
Timing[MaximumClique[t];]
```

```
{0.828 Second, Null}
```

In my view, without significant repair work, this function is pretty much useless. It cannot even handle a 20−vertexrandom graph, gracefully!

CliqueQ

```
?CliqueQ
```

```
CliqueQ[g, c] yields True if the
  list of vertices c defines a clique in graph g.
```

```
CliqueQ[CompleteGraph[12], Range[12]]
```

```
True
```

```
CliqueQ[CompleteGraph[12], Range[5]]
```

```
True
```

```
CliqueQ[RandomGraph[10, .5], Range[5]]
```

```
False
```

MinimumVertexCover

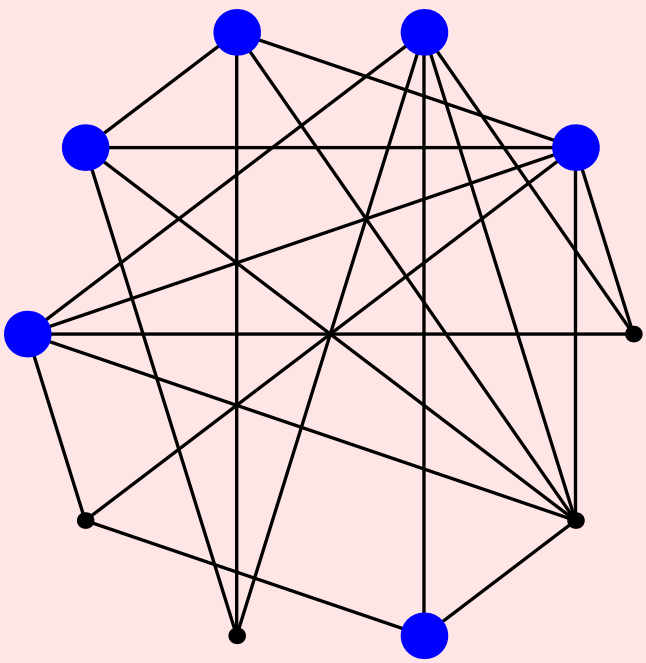
```
?MinimumVertexCover
```

```
MinimumVertexCover[g] finds the minimum vertex cover of graph g.
```

```
?Highlight
```

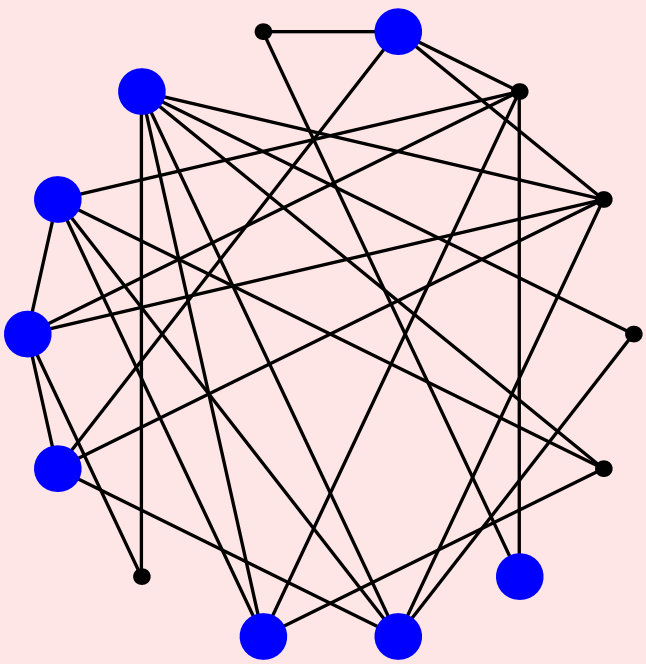
```
Highlight[g, p] displays g with elements in p highlighted. The
  second argument p has the form {s1, s2,...}, where the si's
  are disjoint subsets of vertices and edges of g. The options,
  HighlightedVertexStyle, HighlightedEdgeStyle, HighlightedVertexColors,
  and HighlightedVertexStyle are used to determine the appearanc
  of the highlighted elements of the graph. The default settings
  of the style options are HighlightedVertexStyle->Disc[Fat] and
  HighlightedEdgeStyle->Fat. The options HighlightedVertexColors
  and HighlightedEdgeColors are both set to {Black, Red, Blue,
  Green, Yellow, Purple, Brown, Orange, Olive, Pink, DeepPink,
  DarkGreen, Maroon, Navy}. The colors are chosen from the palette
  of colors with color 1 used for s1, color 2 used for s2, and
  so on. If there are more parts than colors, then the colors
  get used cyclically. The function permits all the options that
  SetGraphOptions permits, for example, VertexColor, VertexStyle,
  EdgeColor, and EdgeStyle. These options can be used to control
  the appearance of the non-highlighted vertices and edges.
```

```
ShowGraph[Highlight[g = RandomGraph[10, .6],
  MinimumVertexCover[g], HighlightedVertexColors → {Blue}]]
```



```
- Graphics -
```

```
ShowGraph[Highlight[g = RandomGraph[14, .3],
  MinimumVertexCover[g], HighlightedVertexColors -> {Blue}]]
```



```
- Graphics -
```

VertexCoverQ

```
?VertexCoverQ
```

```
VertexCoverQ[g, c] yields True if the
  vertices in list c define a vertex cover of graph g.
```

```
MinimumVertexCover[GridGraph[4, 4]]
```

```
$Aborted
```

```
c = {2, 4, 5, 7, 10, 12, 13, 15}
```

```
{2, 4, 5, 7, 10, 12, 13, 15}
```

```
VertexCoverQ[GridGraph[4, 4], c]
```

```
True
```

```
VertexCoverQ[GridGraph[4, 4], Complement[c, {2}]]
```

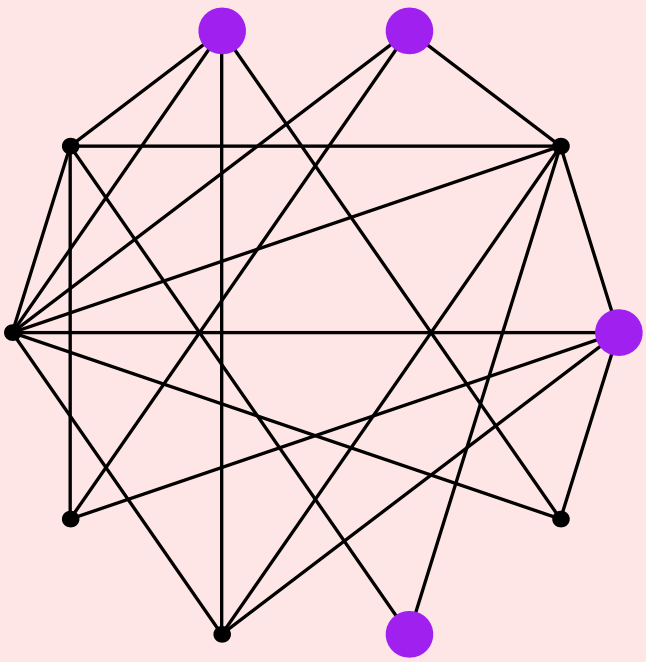
```
False
```

MaximumIndependentSet

```
?MaximumIndependentSet
```

```
MaximumIndependentSet[g] finds the largest independent set of graph g.
```

```
ShowGraph[Highlight[g = RandomGraph[10, .4],
  MaximumIndependentSet[g], HighlightedVertexColors -> {Purple}]]
```



```
- Graphics -
```

These algorithms for vertex cover, independent set, clique etc. are pretty useless. The need is for significant improvements in these algorithms, for simple and possibly randomized heuristics that provide good approximation etc.

IndependentSetQ

```
? IndependentSetQ
```

```
IndependentSetQ[g, i] yields True if the
  vertices in list i define an independent set in graph g.
```

```
IndependentSetQ[CompleteGraph[3, 3], {1, 2, 3}]
```

```
True
```

```
IndependentSetQ[CompleteGraph[3, 3], {3, 4}]
```

```
False
```

PerfectQ

PerfectQ is definitely useless as it is. It cannot even compute whether a 8−vertexgraph is Perfect or not. No point in having this function around.