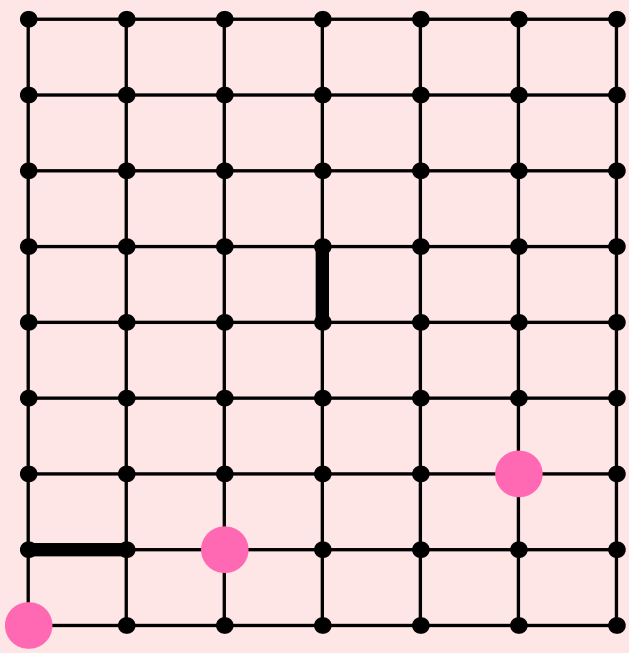
4. Generating Graphs

```
t = GridGraph[7, 9];
```

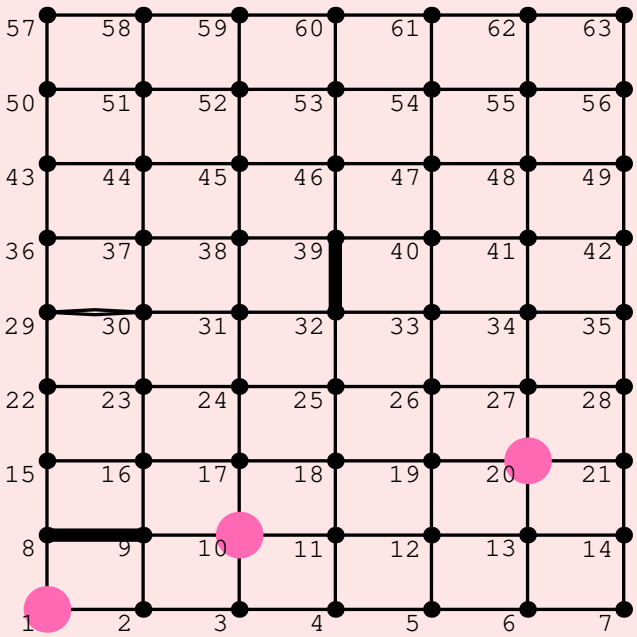
```
ShowGraph[t = SetGraphOptions[t,
   {{1, 10, 20, VertexStyle -> Disc[Large], VertexColor -> HotPink},
    {{8, 9}, {32, 39}, EdgeStyle → Fat}}]]
```



```
- Graphics -
```

```
t = AddEdges[t, {{ {29, 30}}}];
```

```
ShowGraph[t, VertexNumber -> On, PlotRange -> Large[0.02]]
```




```
- Graphics -
```

GraphUnion

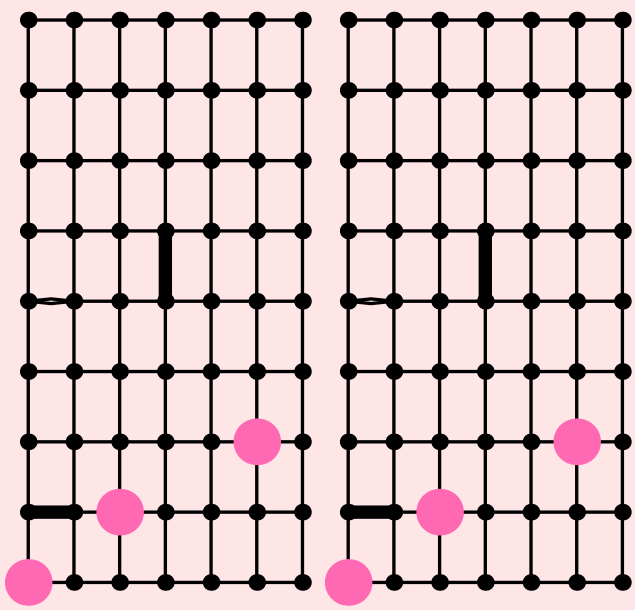
```
?GraphUnion
```

```
GraphUnion[g1, g2, ...] constructs the union of
  graphs g1, g2, and so forth. GraphUnion[n, g] constructs
  n copies of graph g, for any non-negative integer n.
```

NOTES

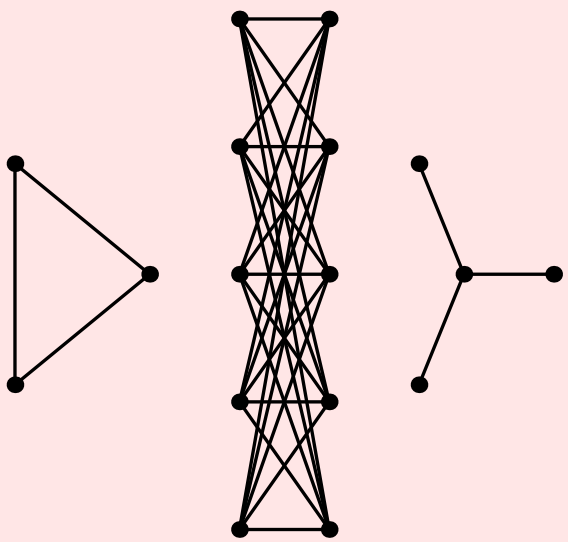
* GraphUnion preserves the graphics options of the two graphs.

```
ShowGraph[GraphUnion[t, t], PlotRange -> Large[0.02]]
```



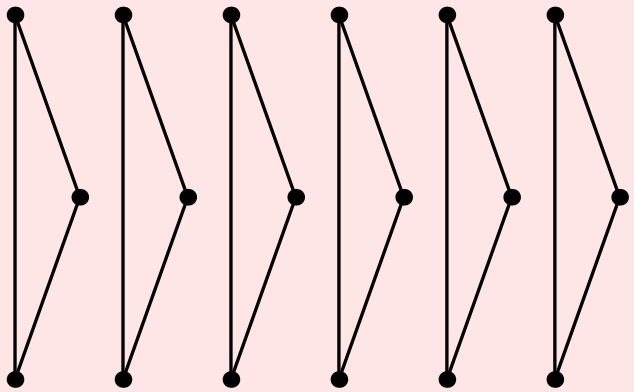
```
- Graphics -
```

```
ShowGraph[GraphUnion[CompleteGraph[3], CompleteGraph[5, 5], Star[4]],
 PlotRange -> Large[0.1]]
```



- Graphics -

```
ShowGraph[ GraphUnion [6, CompleteGraph[3]]]
```



```
- Graphics -
```

TIMING DISCUSSION

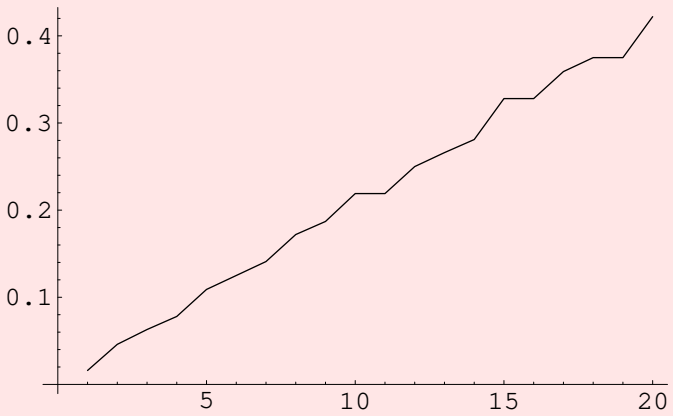
GraphUnion should be extremely fast and asympototically linear in the number of edges + vertices in the resulting graph.

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

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

```
{{0.016 Second, Null}, {0.046 Second, Null},
 {0.063 Second, Null}, {0.078 Second, Null}, {0.109 Second, Null},
 {0.125 Second, Null}, {0.141 Second, Null}, {0.172 Second, Null},
 {0.187 Second, Null}, {0.219 Second, Null}, {0.219 Second, Null},
 {0.25 Second, Null}, {0.266 Second, Null}, {0.281 Second, Null},
 {0.328 Second, Null}, {0.328 Second, Null}, {0.359 Second, Null},
 {0.375 Second, Null}, {0.375 Second, Null}, {0.422 Second, Null}}
```

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




```
- Graphics -
```

Here is a comparison between the old and the new implementations of GraphUnion. The improvement in time is significant.

```
a = DiscreteMath`OldCombinatorica`Wheel[200]; aa = Wheel[200];
```

```
b = DiscreteMath`OldCombinatorica`CompleteGraph[75]; bb = CompleteGraph[75];
```

```
{Timing[DiscreteMath`OldCombinatorica`GraphUnion[a, b];],
 Timing[GraphUnion[aa, bb];]}
```

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

```
cc = Star[1000]; c = DiscreteMath`OldCombinatorica`Star[1000];
```

```
Timing[ GraphUnion[aa, cc];]
```

```
{0.047 Second, Null}
```

```
Timing[ DiscreteMath`OldCombinatorica`GraphUnion[a, c]; ]
```

```
{6.484 Second, Null}
```

ExpandGraph

```
?ExpandGraph
```

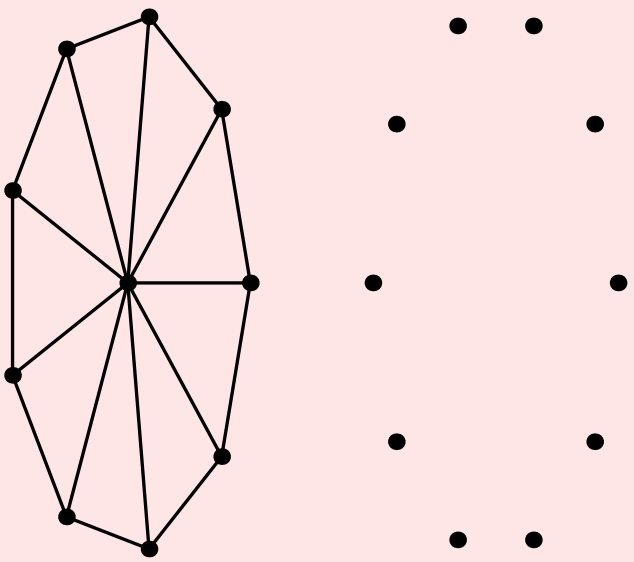
```
ExpandGraph[g, n] expands graph g to n vertices by adding disconnected
  vertices. This is obsolete. Use AddVertices[g, n] instead.
```

I don't see the need for ExpandGraph. AddVertices can replace ExpandGraph.

```
?AddVertices
```

```
AddVertices[g, n] adds n disconnected vertices to graph g. AddVertices[
  g, vList] adds vertices in  vList to g. vList contains embedding
  and graphics information and can have the form {x, y} or {{x1, y1},
  {x2, y2}...} or the form {{{x1, y1}, g1}, {{x2, y2}, g2},...},
  where {x, y}, {x1, y1}, and {x2, y2} are point coordinates and
  g1 and g2 are graphics information associated with vertices.
```

```
ShowGraph[ExpandGraph[Wheel[10], 20]]
```



```
- Graphics -
```

GraphIntersection

```
?GraphIntersection
```

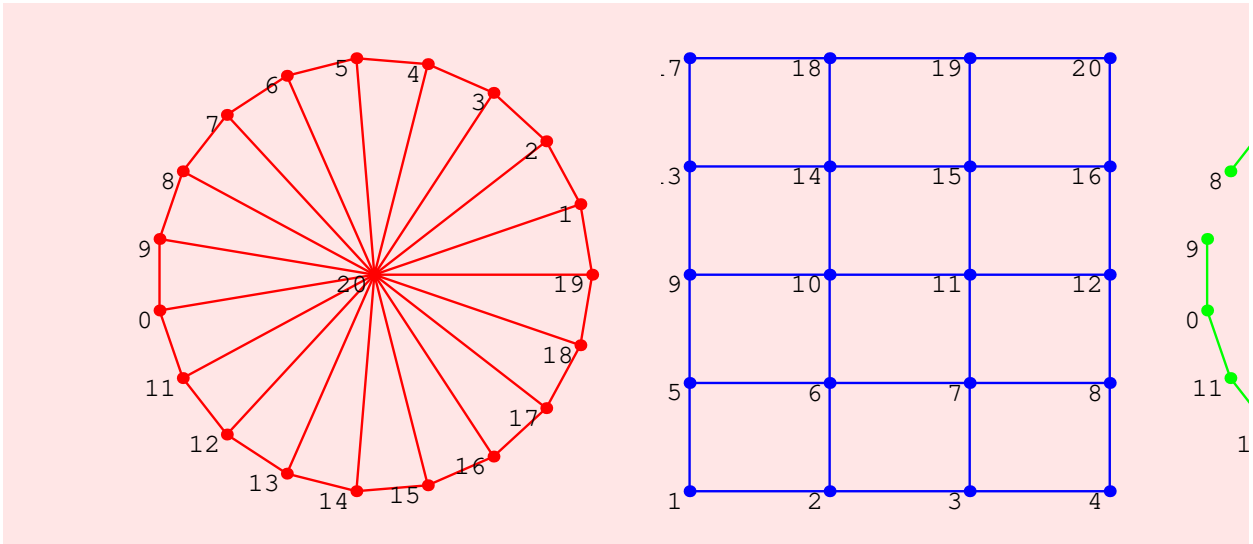
```
GraphIntersection[g1, g2, ...] constructs the graph
  defined by the edges that are in all the graphs g1, g2, ....
```

```
p1 = ShowGraph[s = Wheel[20], VertexColor -> Red,
   EdgeColor -> Red, VertexNumber -> On, Graphics];
```

```
p2 = ShowGraph[r = GridGraph[4, 5], VertexColor -> Blue,
   EdgeColor -> Blue, VertexNumber -> On, Graphics];
```

```
p3 = ShowGraph[GraphIntersection[s, r], VertexColor -> Green,
   EdgeColor -> Green, VertexNumber -> On, Graphics];
```

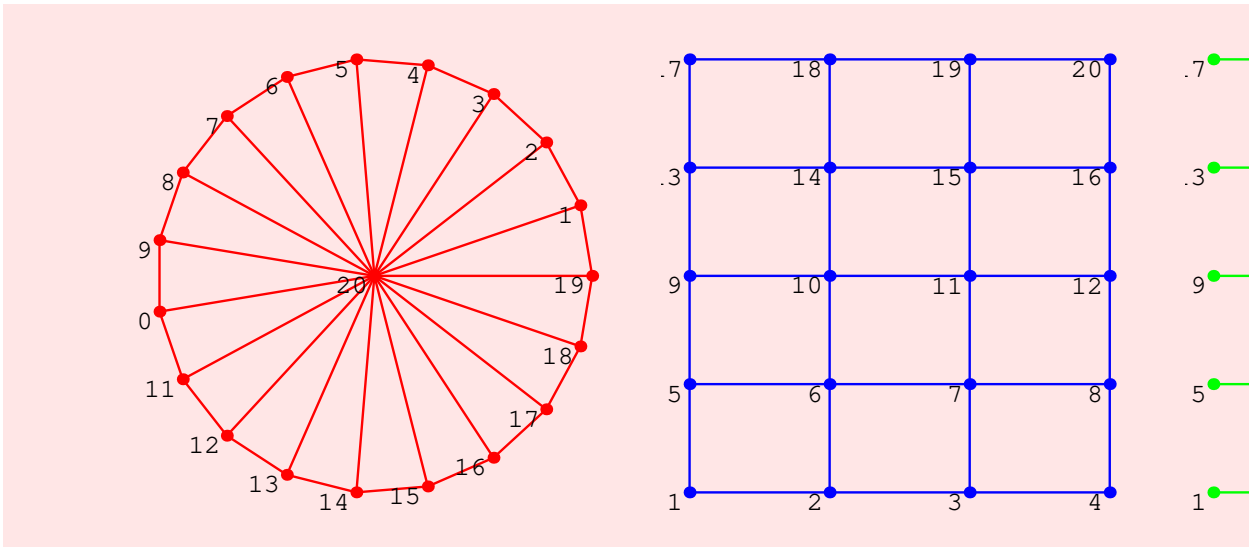
```
Show[ GraphicsArray[{p1, p2, p3} ], ImageSize -> 600]
```




```
- GraphicsArray -
```

```
p4 = ShowGraph[GraphIntersection[r, s], VertexColor -> Green,
   EdgeColor -> Green, VertexNumber -> On, Graphics];
```

```
Show[ GraphicsArray[{p1, p2, p4} ], ImageSize -> 600]
```

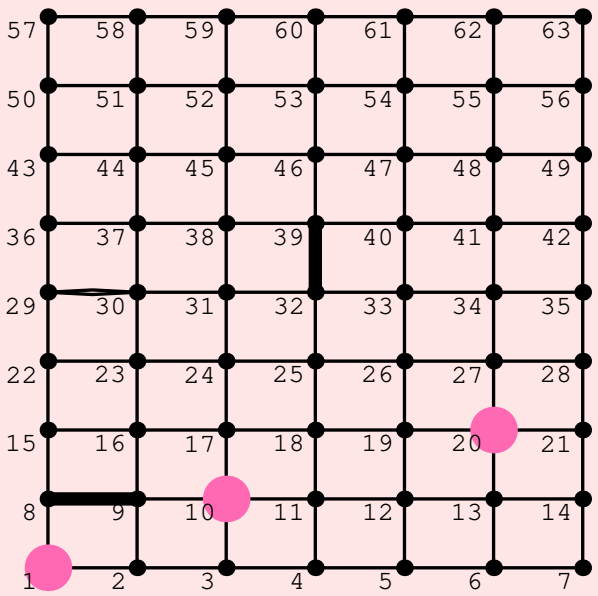



```
- GraphicsArray -
```

NOTES

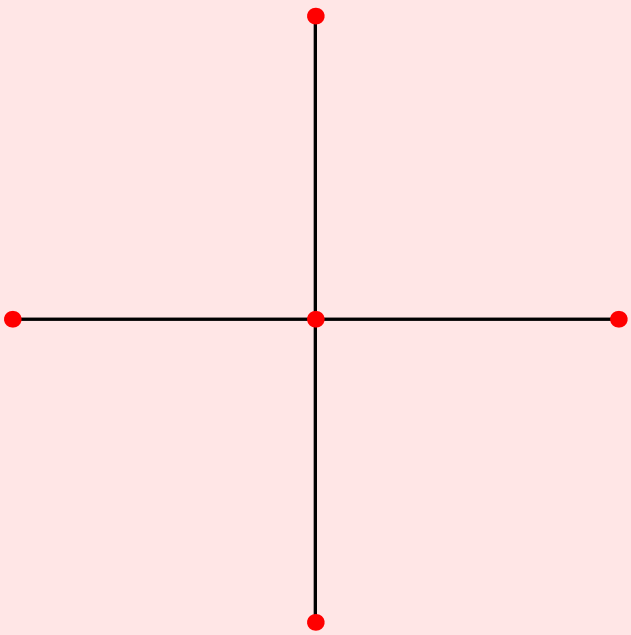
*There is an asymmetry in the function as far as inheritance of graphics information is concerned. The edge and vertex graphics information of the first graph are inherited by the result. Users should keep this in mind while choosing the order in which to specify the graphs.

```
ShowGraph[GraphIntersection[t, CompleteGraph[63]],
 VertexNumber -> On, PlotRange -> Large[0.1]]
```



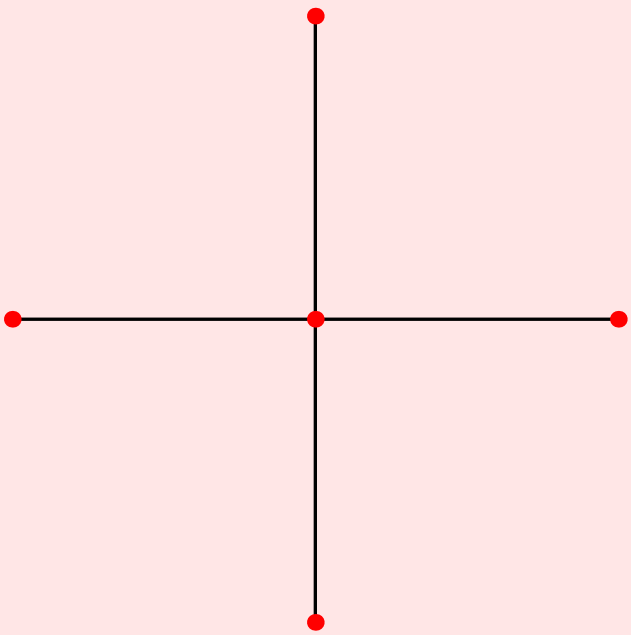

```
- Graphics -
```

```
g = SetGraphOptions[Star[5], VertexColor → Red]; ShowGraph[g]
```



```
- Graphics -
```

```
ShowGraph[GraphIntersection[g, CompleteGraph[5]]]
```



```
- Graphics -
```

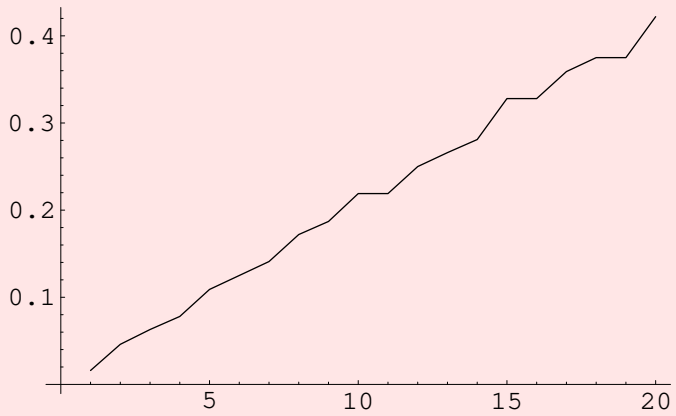
TIMING DISCUSSION

Let g1 be an m1−vertexgraph and g2 be an m2−vertexgraph. Let the number of edges in g1 ⋂ g2 be d. Then the asymptotic running time of the GraphIntersection function is θ(m1 log (m1) + m2 log(m2) + m1 d). Assuming that g1 and g2 are identical, this leads to a function that is asymptotic in m, the number of edges in the graph. Let us see if these claims bear out in experiments.

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

```
$Aborted
```

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




```
- Graphics -
```

The plot above is clearly quadratic and this brings up the question whether GraphIntersection could be rewritten so that it is sub−quadraticin the number of edges.

Here is a timing comparison between the new and the old implementations of GraphIntersection. For dense graphs the old implementation seems to be better, whereas for sparse graphs the new implementation of graph intersection does better.

```
a = DiscreteMath`OldCombinatorica`CompleteGraph[80]; aa = CompleteGraph[80];
```

```
b = DiscreteMath`OldCombinatorica`GridGraph[8, 10]; bb = GridGraph[8, 10];
```

```
{Timing[DiscreteMath`OldCombinatorica`GraphIntersection[a, b];],
 Timing[GraphIntersection[aa, bb];], Timing[ GraphIntersection[bb, aa];]}
```

```
{{0.078 Second, Null}, {0.375 Second, Null}, {0.016 Second, Null}}
```

```
a = DiscreteMath`OldCombinatorica`RandomTree[100]; aa = RandomTree[100];
```

```
b = DiscreteMath`OldCombinatorica`GridGraph[10, 10]; bb = GridGraph[10, 10];
```

```
{Timing[DiscreteMath`OldCombinatorica`GraphIntersection[a, b];],
 Timing[GraphIntersection[aa, bb];]}
```

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

GraphDifference

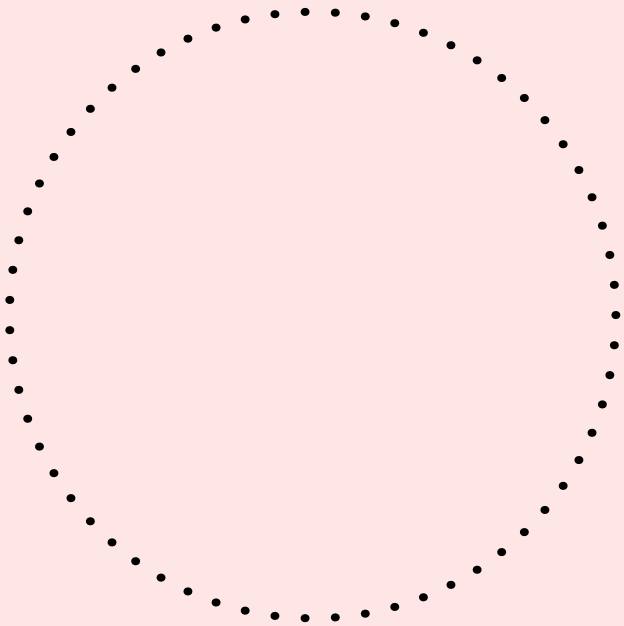
```
? GraphDifference
```

```
GraphDifference[g, h] constructs the graph resulting from
  subtracting the edges of graph h from the edges of graph g.
```

```
Edges[GraphDifference[t, t]]
```

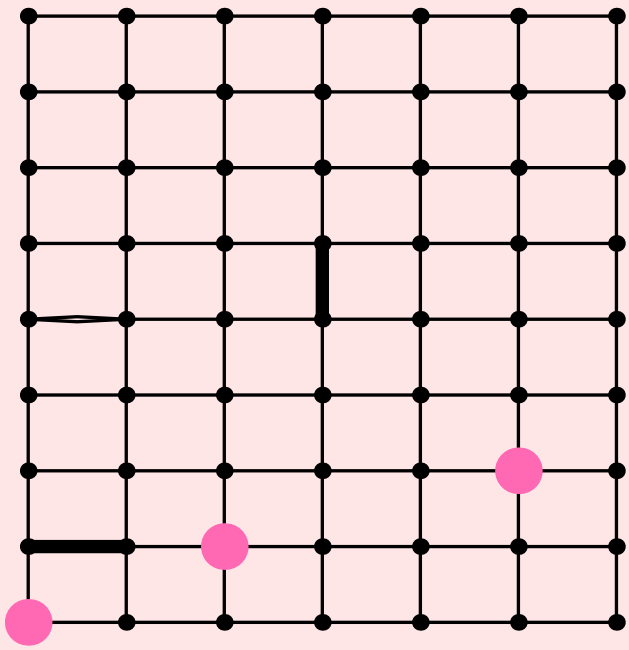
```
{}
```

```
ShowGraph[s = EmptyGraph[V[t]], VertexStyle -> Disc[Small]]
```



```
- Graphics -
```

```
ShowGraph[GraphDifference[t, s]]
```



```
- Graphics -
```

NOTES

Again, graphics information from the first argument is inherited.

TIMING DISCUSSION

The code for GraphDifference is very similar to the code for GraphIntersection. So I expect that experiments will show that the function has a similar worst case quadratic behaviour. However, the old implementation of GraphDifference is faster than the new implementation. It is impossible to beat GraphDifference since GraphDifference is simply matrix addition.

```
a = DiscreteMath`OldCombinatorica`CompleteGraph[100]; aa = CompleteGraph[100]
```

```
b = DiscreteMath`OldCombinatorica`RandomTree[100]; bb = RandomTree[100];
```

```
{Timing[DiscreteMath`OldCombinatorica`GraphDifference[a, a];],
 Timing[GraphDifference[aa, aa];]}
```

```
{{0.031 Second, Null}, {0.062 Second, Null}}
```

```
{Timing[DiscreteMath`OldCombinatorica`GraphDifference[b, b];],
 Timing[GraphDifference[bb, bb];]}
```

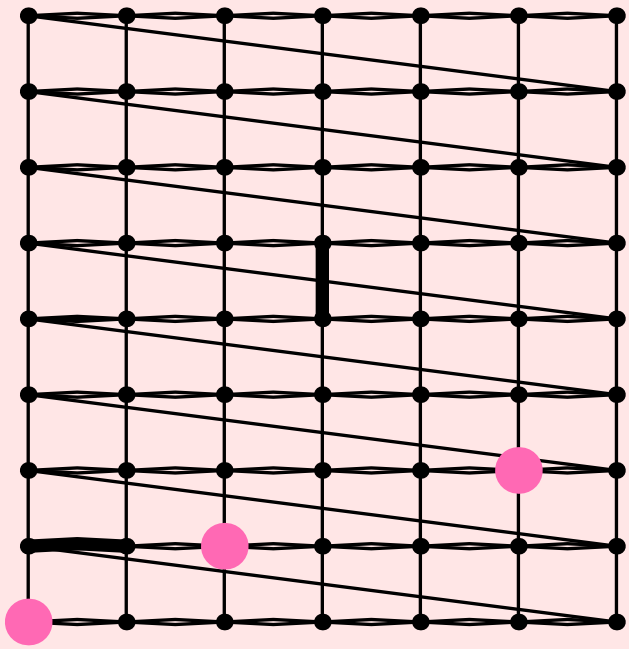
```
{{0.031 Second, Null}, {0. Second, Null}}
```

GraphSum

```
?GraphSum
```

```
GraphSum[g1, g2, ...] constructs the graph resulting
  from joining the edge lists of graphs g1, g2, and so forth.
```

```
ShowGraph[GraphSum[t, Path[63]]]
```

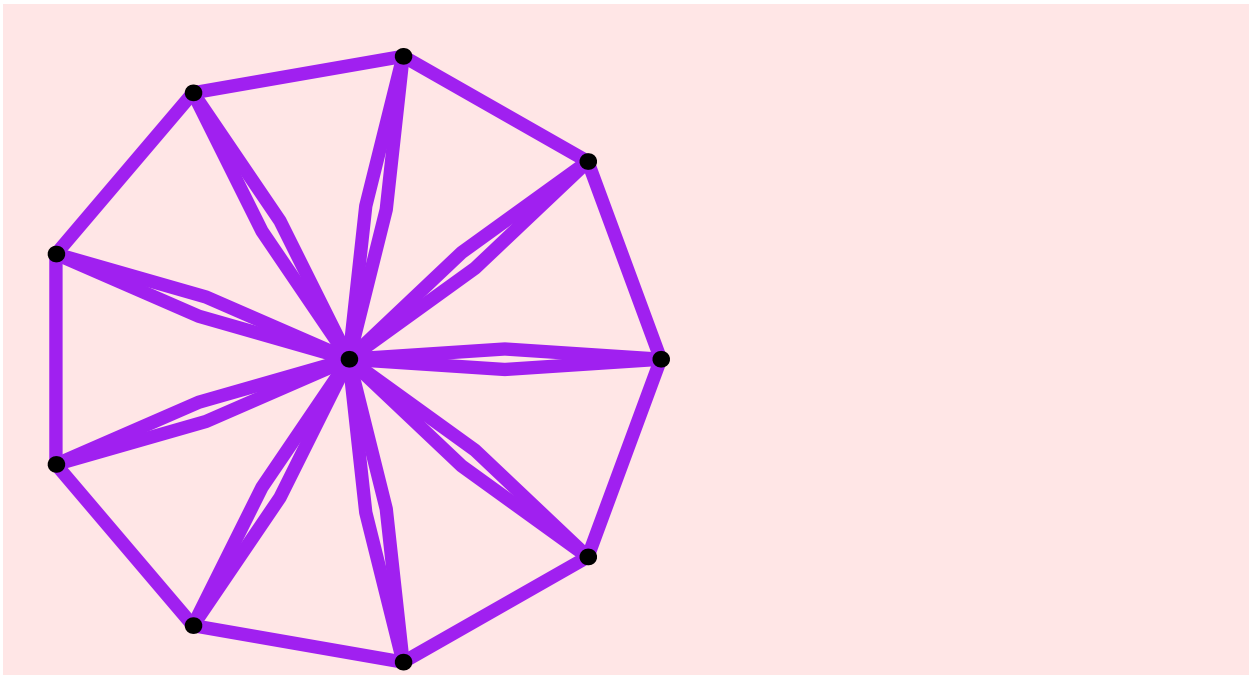


```
- Graphics -
```

```
g = SetGraphOptions[Wheel[10], EdgeStyle -> Fat, EdgeColor -> Purple];
```

```
h = SetGraphOptions[Star[10], EdgeColor → Green];
```

```
ShowGraph[GraphSum[g, Star[10]]]
```



```
- Graphics -
```

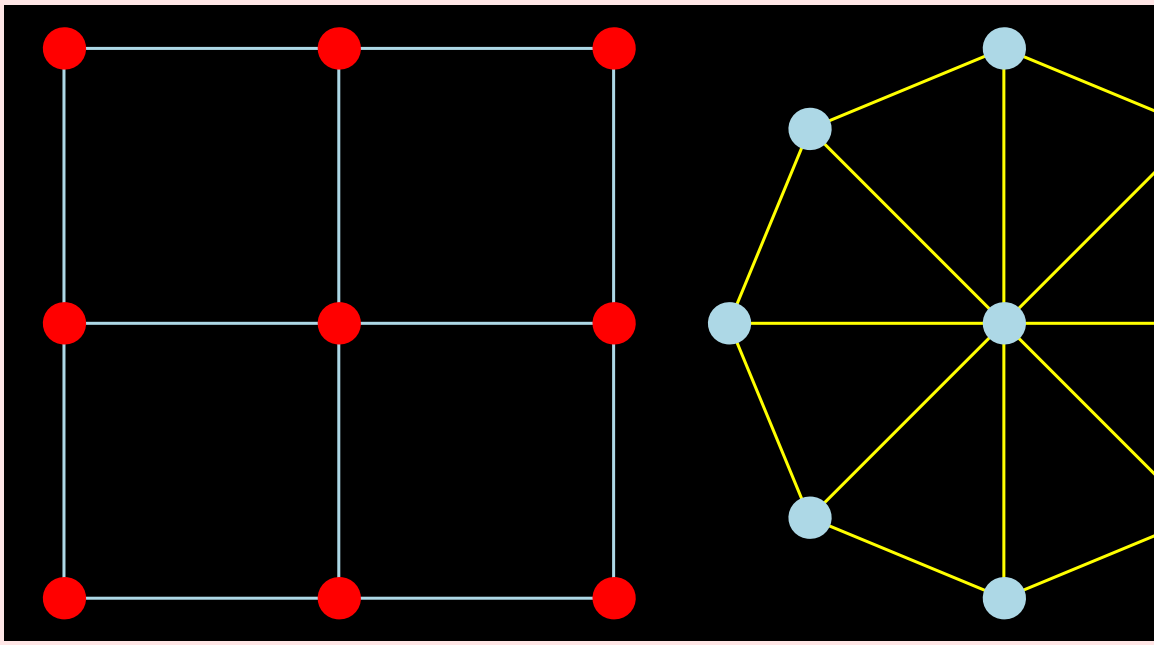
BUG

In the above example, the edges of Star[10] should keep their graphics properties and not inherit the graphics properties of the first graph. The following example also shows the same problem. I will fix this when I work on the graph data structure.

```
p1 = ShowGraph[a = SetGraphOptions[ GridGraph[3, 3], EdgeColor -> LightBlue,
     VertexColor -> Red, VertexStyle -> Disc[Large]], Graphics];
```

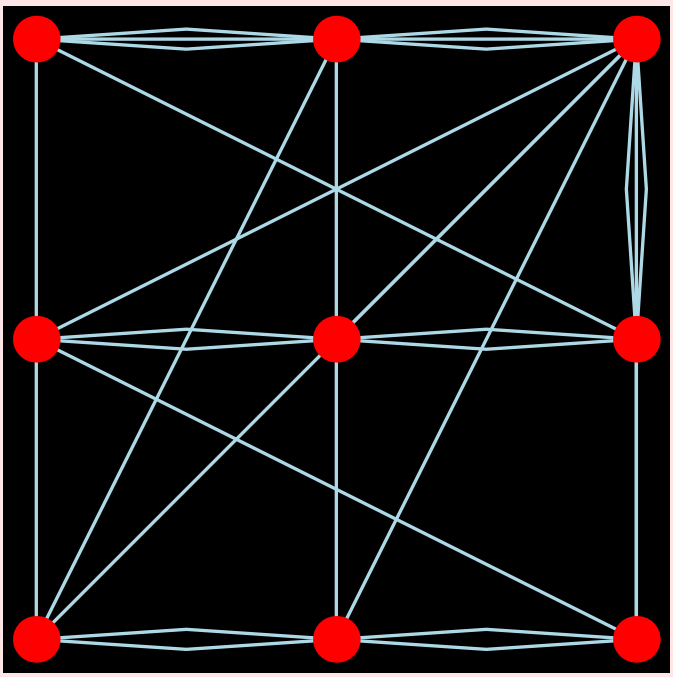
```
p2 = ShowGraph[b = SetGraphOptions[ Wheel[9], EdgeColor -> Yellow,
     VertexColor -> LightBlue, VertexStyle -> Disc[Large]], Graphics];
```

```
Show[GraphicsArray[{p1, p2}], ImageSize -> 500, Background -> Black]
```



```
- GraphicsArray -
```

```
ShowGraph[ GraphSum[ a, b ], Background -> Black , PlotRange -> Large[0.01]]
```

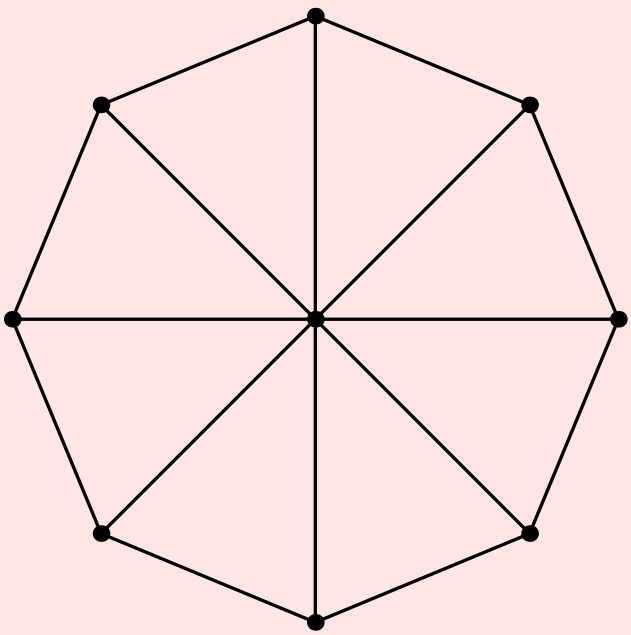


```
- Graphics -
```

What happens if the graphics information in the second graph is local? The following example shows that edge–graphics information is inherited correctly in this case.

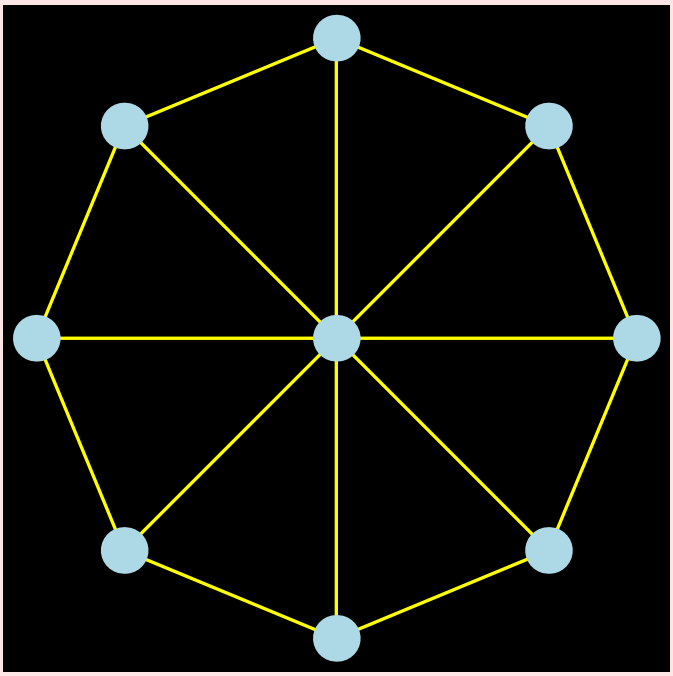
```
b = Wheel[9];
```

```
ShowGraph[b]
```



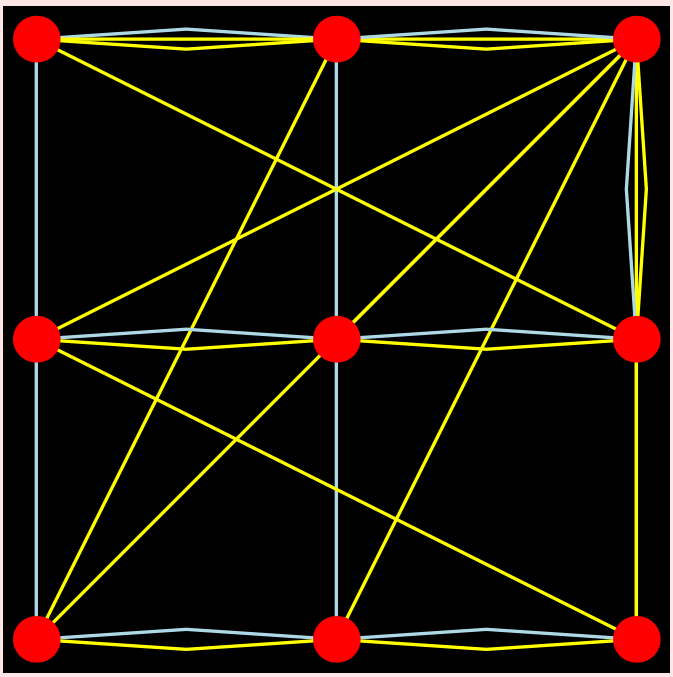
```
- Graphics -
```

```
p2 = ShowGraph[b = SetGraphOptions[ b, Append[Edges[b], EdgeColor -> Yellow],
    VertexColor -> LightBlue, VertexStyle -> Disc[Large]],
  Background -> Black, PlotRange -> Large[0.01]]
```



```
- Graphics -
```

```
ShowGraph[ GraphSum[ a, b ], Background -> Black , PlotRange -> Large[0.01]]
```



```
- Graphics -
```

TIMING DISCUSSION

GraphSum is simply a join of the edge lists and for all practical purposes should take O(1) time. Whether this is really true depends on the Join[...] operation. In any case, this function is extremely fast as revealed by the experiments below. Given how fast the function is, the plot below does not make a whole lot of sense.

The old implementation of this function is also quite fast since it involves adding the adjacency matrices.

```
gt = Table[ GridGraph[20, 10 i], {i, 20}];
rt = Table[Timing[GraphSum [ gt[[i]], 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}, {0.015 Second, Null},
 {0.016 Second, Null}, {0.031 Second, Null}, {0.032 Second, Null},
 {0.031 Second, Null}, {0.047 Second, Null}, {0.031 Second, Null},
 {0.047 Second, Null}, {0.047 Second, Null}, {0.047 Second, Null},
 {0.062 Second, Null}, {0.063 Second, Null}, {0.046 Second, Null}}
```

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

```
- Graphics -
```

```
aa = Table[ DiscreteMath`OldCombinatorica`RandomTree[i^2], {i, 10, 30, 10}];
```

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

```
Table [
 Timing[ DiscreteMath`OldCombinatorica`GraphSum[aa[[i]], bb[[i]] ];], {i, 3}]
```

```
{{0. Second, Null}, {0.016 Second, Null}, {0.219 Second, Null}}
```

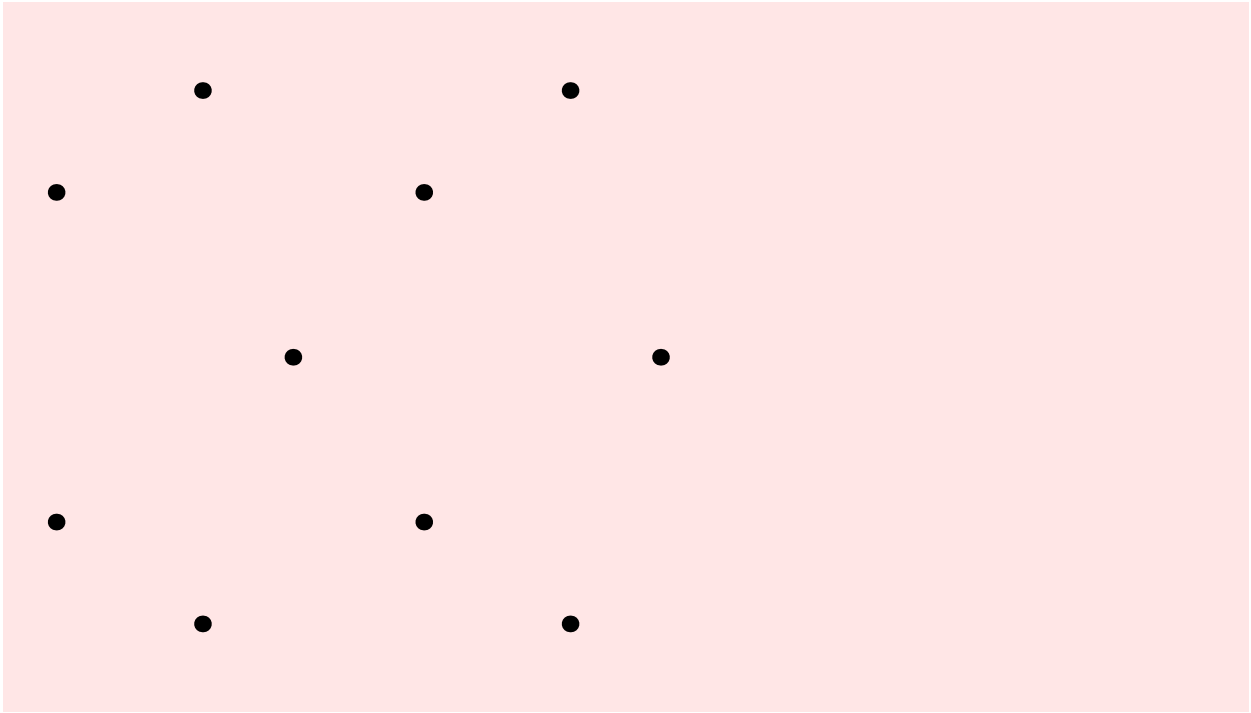
GraphJoin

```
? GraphJoin
```

```
GraphJoin[g1, g2, ...] constructs the join of graphs g1, g2, and
  so on. This is the graph obtained by adding all possible edges
  between different graphs to the graph union of g1, g2, ....
```

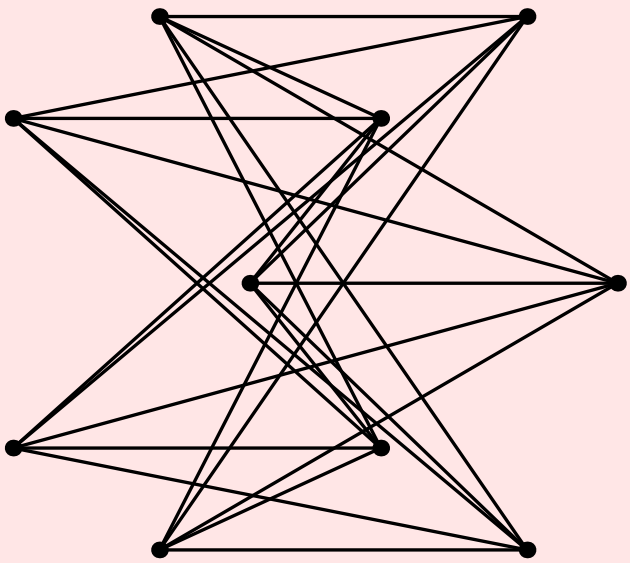
```
l = r = EmptyGraph[5];
```

```
ShowGraph[GraphUnion[l, r]]
```



```
- Graphics -
```

```
ShowGraph[GraphJoin[l, r]]
```



```
- Graphics -
```

BUG

The same graphics inheritance problem continues. Should the vertex in g2 loose its Green color? If the Green color were specified as a local option, then this information is not lost (see example further below). Should making the right choice be the users burden?

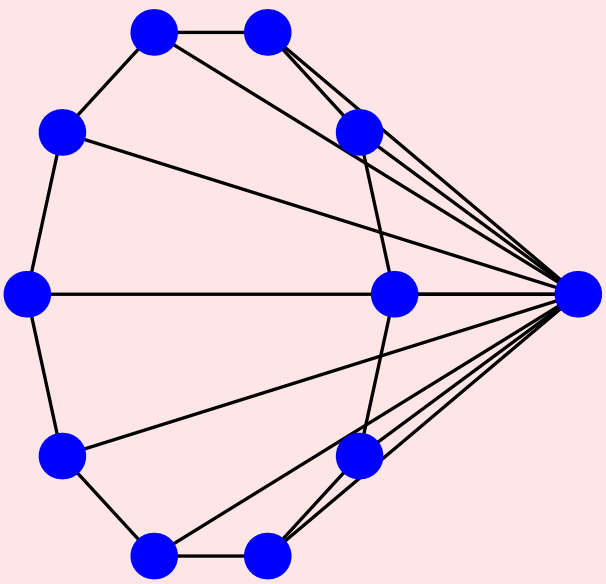
The principle is that as much as possible graphics information is maintained from the original graphs. This means that no, g2's vertices should not loose their green color.

```
g1 =
  SetGraphOptions[ Cycle[10], VertexStyle -> Disc[Large], VertexColor -> Blue];
```

```
g2 = SetGraphOptions[CompleteGraph[1],
   VertexColor -> Green, VertexStyle -> Disc[Large]];
```

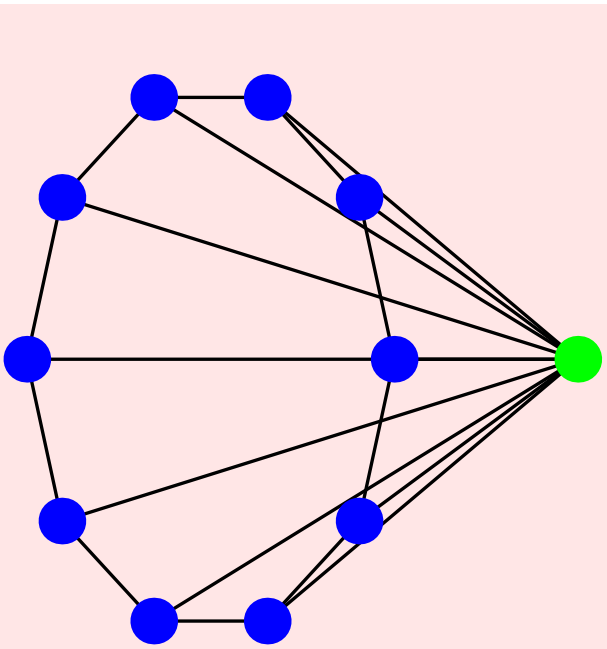
```
g3 = SetGraphOptions[CompleteGraph[1],
   {1, VertexColor -> Green}, VertexStyle -> Disc[Large]];
```

```
ShowGraph[GraphJoin[g1, g2], PlotRange -> Large[0.1]]
```



```
- Graphics -
```

```
ShowGraph[ GraphJoin[g1, g3], PlotRange -> Large[0.1]]
```



```
- Graphics -
```

TIMING DISCUSSION

The running time of the function is θ(n1n2+m1+m2) where n1 and m1 are the number of vertices and edges in graph 1 and n1 and m2 are the number of vertices and edges in graph 2. So the running time is quadratic in the size of the input. This is confirmed by the following experiment.

Further below there is a comparison of the running time of the new implementation and the running time of the old implementation. The new implementation is substantially faster by virtue of the sparse data structure it uses.

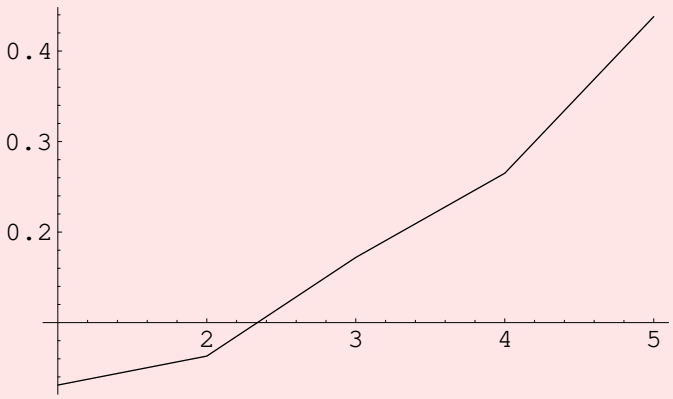
```
? GraphJoin
```

```
GraphJoin[g1, g2, ...] constructs the join of graphs g1, g2, and
  so on. This is the graph obtained by adding all possible edges
  between different graphs to the graph union of g1, g2, ....
```

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

```
rt = Table[Timing[GraphJoin[ gt[[i]], gt[[i]]];], {i, 5}];
```

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




```
- Graphics -
```

```
a = Table[DiscreteMath`OldCombinatorica`RandomTree[i], {i, 50, 70, 10}];
aa = Table[RandomTree[i], {i, 50, 70, 10}];
```

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

```
Table[
 Timing[DiscreteMath`OldCombinatorica`GraphJoin[a[[i]], b[[i]] ];], {i, 3}]
```

```
{{0.328 Second, Null}, {4.812 Second, Null}, {22.579 Second, Null}}
```

```
Table[ Timing[GraphJoin[aa[[i]], bb[[i]] ];], {i, 2}]
```

```
{{0.125 Second, Null}, {0.64 Second, Null}}
```

GraphProduct

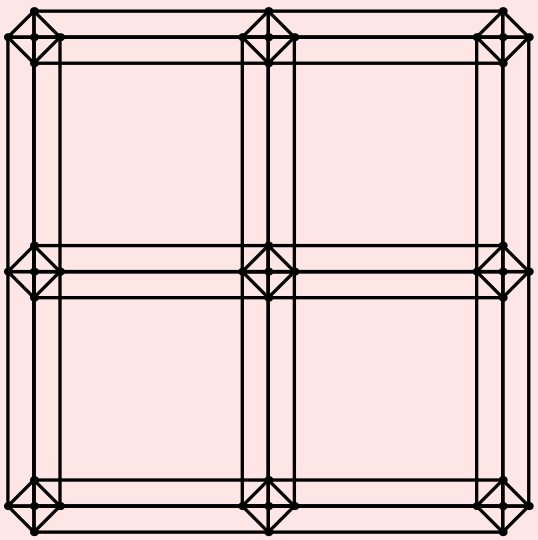
```
? GraphProduct
```

```
GraphProduct[g1, g2, ...] constructs
  the product of graphs g1, g2, and so forth.
```

```
l = Wheel[5];
```

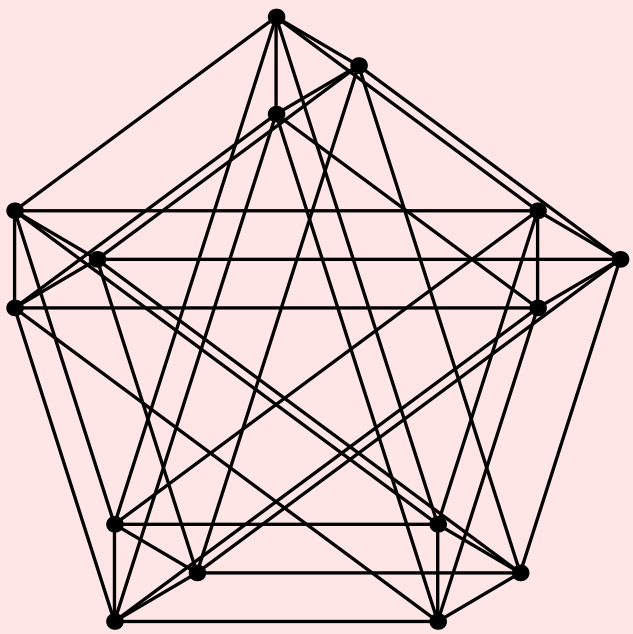
```
r = GridGraph[3, 3];
```

```
ShowGraph[GraphProduct[l, r], VertexStyle -> Disc[Small]]
```



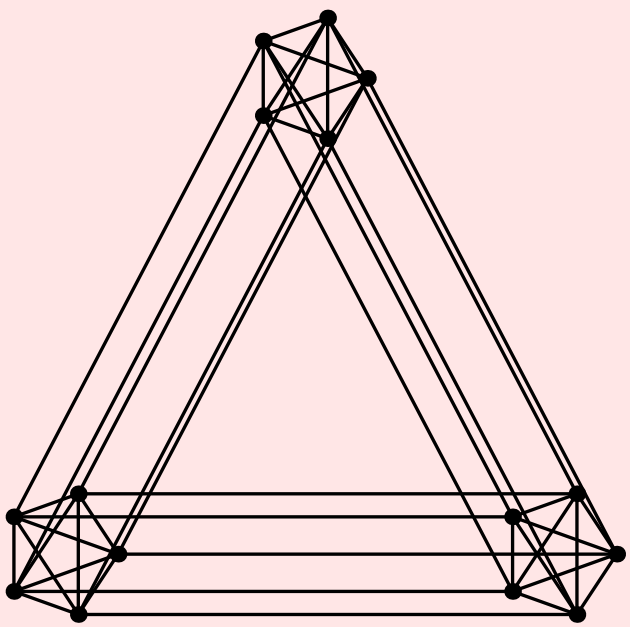
```
- Graphics -
```

```
ShowGraph[GraphProduct[CompleteGraph[3], CompleteGraph[5]]]
```



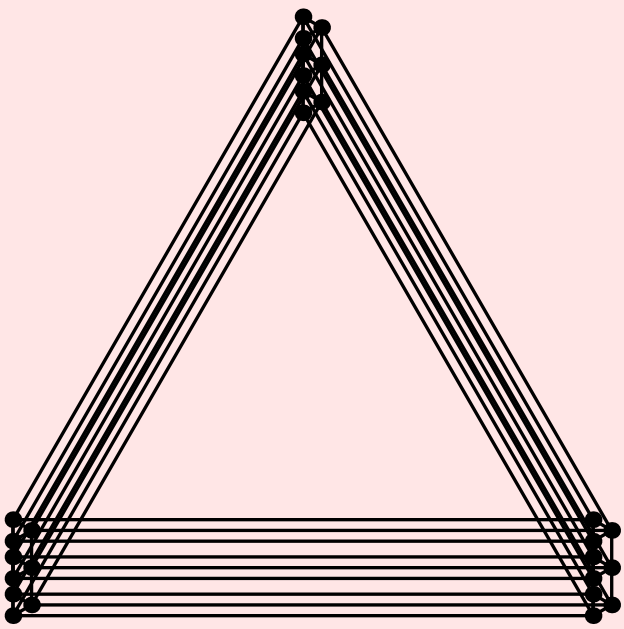
```
- Graphics -
```

```
ShowGraph[GraphProduct[CompleteGraph[5], CompleteGraph[3]]]
```



```
- Graphics -
```

```
ShowGraph[ GraphProduct[CompleteGraph[3], Star[3], CompleteGraph[3]]]
```

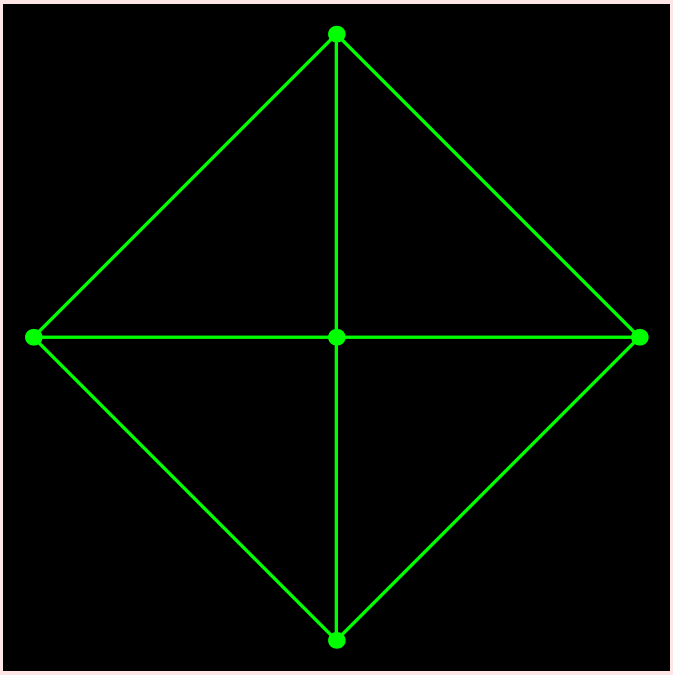


```
- Graphics -
```

BUG

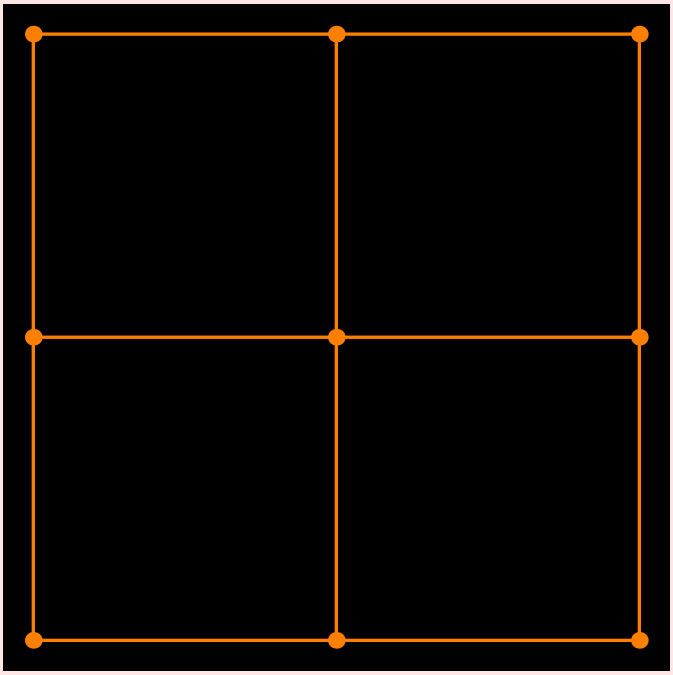
As with GraphSum etc. Graphics information that is local is not inherited. This is shown in the following example. In fact, here graphics information from both graphs is missing. So in that sense this is a bug peculiar to GraphProduct.

```
ShowGraph[l = SetGraphOptions[l, VertexColor -> Green, EdgeColor -> Green],
 Background -> Black]
```



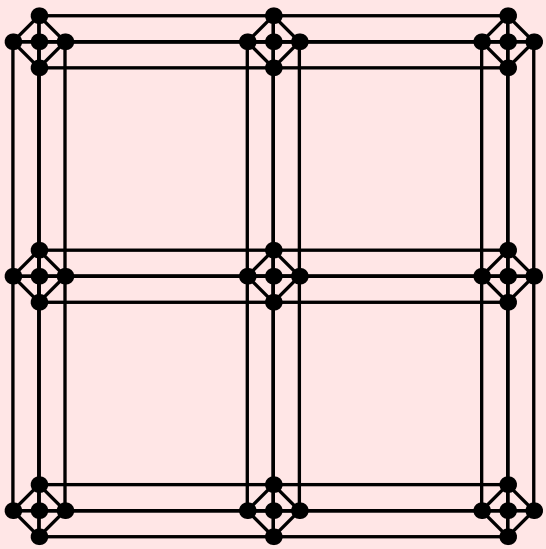
```
- Graphics -
```

```
ShowGraph[r = SetGraphOptions[r, VertexColor -> Orange, EdgeColor -> Orange],
 Background -> Black]
```



```
- Graphics -
```

```
ShowGraph[GraphProduct[l, r]]
```



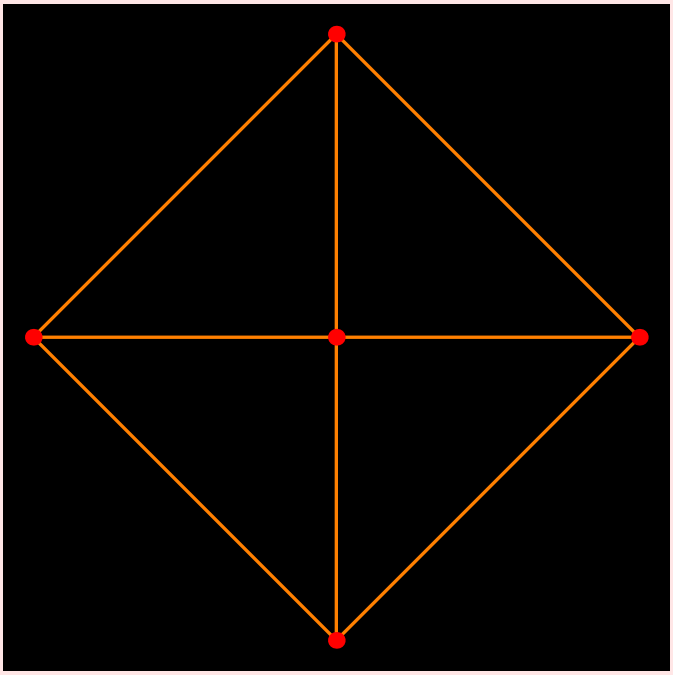
```
- Graphics -
```

BUG

There is a problem with inheriting local graphics information associated with vertices. The following example shows this. But, it not clear what convention to use for the vertices of the resulting graph. Something to think about.

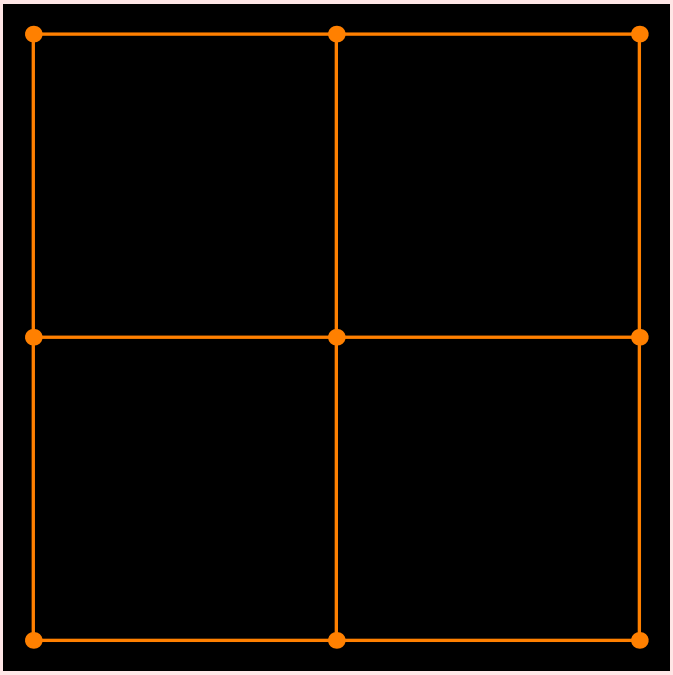
The many copies of g1 should inherit graphics information of g1 and the edges that connect these copies should inherit information from g2.

```
ShowGraph[l1 = SetGraphOptions[l, VertexColor → Red, EdgeColor → Orange],
 Background → Black]
```



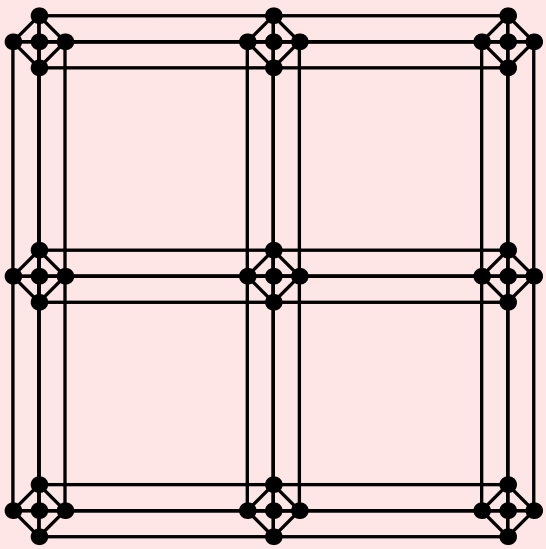
```
- Graphics -
```

```
ShowGraph[r1 = SetGraphOptions[r, VertexColor -> Orange, EdgeColor -> Orange],
 Background -> Black]
```



```
- Graphics -
```

```
ShowGraph[GraphProduct[l, r]]
```



```
- Graphics -
```

TIMING DISCUSSION

I need to look at the GraphProduct code somewhat carefully to figure out the asymptotic running time of the function. However, the output size if quadratic in the input size and I suspect the running time of the function is no worse than the size of the output produced. The following experiment bears this conjecture out. What is remarkable is that in under 22 seconds, the function could produce a graph with 62,500 vertices and 222,500 edges! This supports our view that now Combinatorica is not just a tool to play with small examples, but can be used for reasonable sized computations.

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

```
{{0.062 Second, Null}, {0.219 Second, Null},
 {0.641 Second, Null}, {1.156 Second, Null},
 {1.781 Second, Null}, {2.547 Second, Null}, {3.516 Second, Null},
 {4.656 Second, Null}, {5.953 Second, Null}, {6.235 Second, Null}}
```

```
g = GraphProduct[ gt[[10]], gt[[10]]];
```

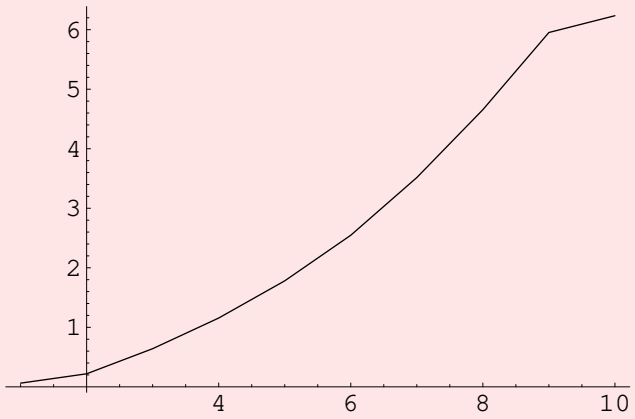
```
M[g]
```

```
222500
```

```
V[g]
```

```
62500
```

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




```
- Graphics -
```

```
aa = Cycle[100];
```

```
bb = RandomTree[100];
```

```
Timing[(ans = GraphProduct[aa, bb]);]
```

```
{0.735 Second, Null}
```

```
V[ans]
```

```
10000
```

```
M[ans]
```

```
19900
```

```
a = DiscreteMath`OldCombinatorica`Cycle[30];
```

```
aa = Cycle[30];
```

```
b = DiscreteMath`OldCombinatorica`RandomTree[30];
```

```
bb = RandomTree[30];
```

```
{Timing[DiscreteMath`OldCombinatorica`GraphProduct[a, b];],
 Timing[GraphProduct[aa, bb];]}
```

```
{{1.484 Second, Null}, {0.078 Second, Null}}
```

NOTES

* Earlier I ran the above experiment with 50 instead of 30 as the argument for the various functions. But, that can occasionally crash my machine because the old implementation of GraphProduct takes up too much memory. When I run MMa 4.1 and run out of memory, the kernel shuts down and I get a polite message that there is no more memory. But, with MMa 3.0, the machine crashes.

IncidenceMatrix

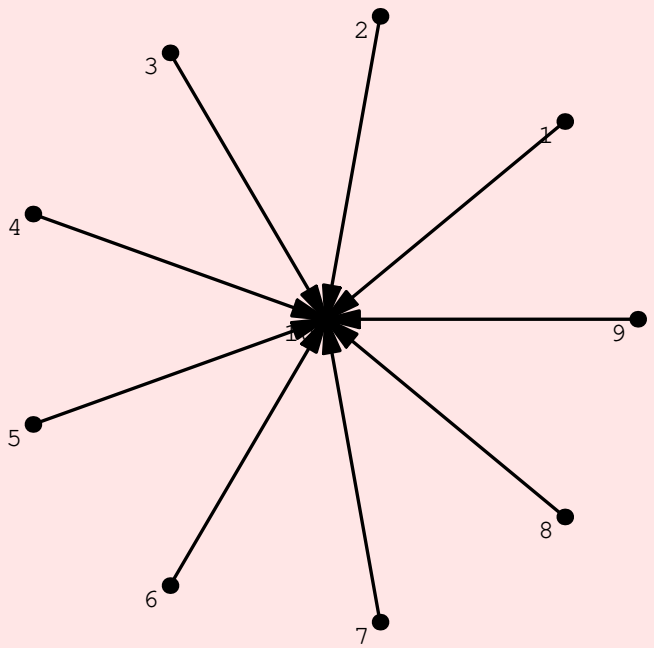
```
?IncidenceMatrix
```

```
IncidenceMatrix[g] returns the (0, 1)-matrix of graph g, which
  has a row for each vertex and a column for each edge and (
  v, e) = 1 if and only if vertex v is incident upon edge e.
  For a directed graph (v, e) = 1 if edge e is outgoing from v.
```

```
MatrixForm[IncidenceMatrix[Star[10]]]
```

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

```
g = SetGraphOptions[ Star[10], EdgeDirection → On];
ShowGraph[g, VertexNumber → On]
```




```
- Graphics -
```

```
MatrixForm[IncidenceMatrix[g]]
```

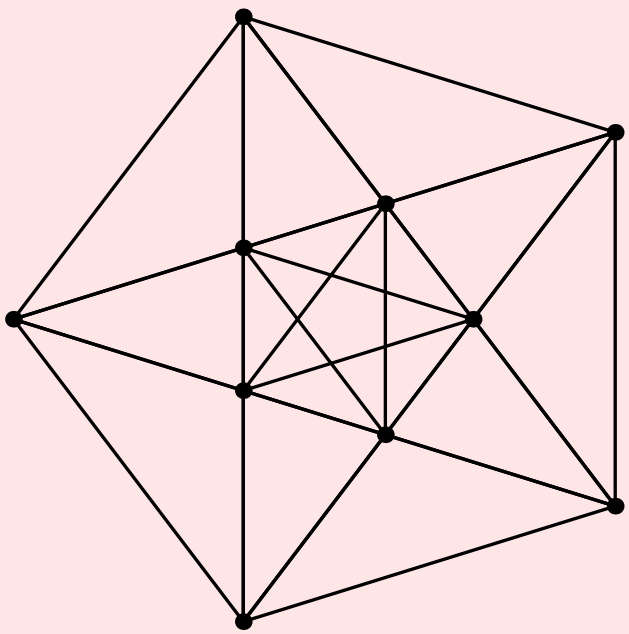
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 &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

LineGraph

```
?LineGraph
```

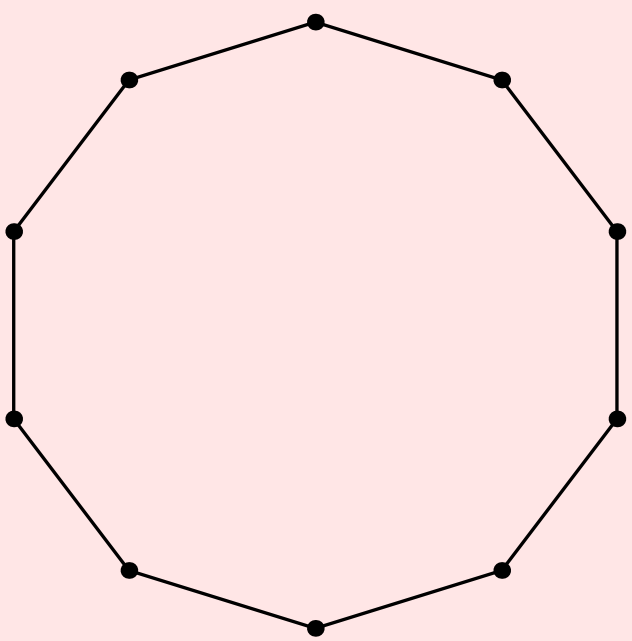
```
LineGraph[g] constructs the line graph of graph g.
```

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



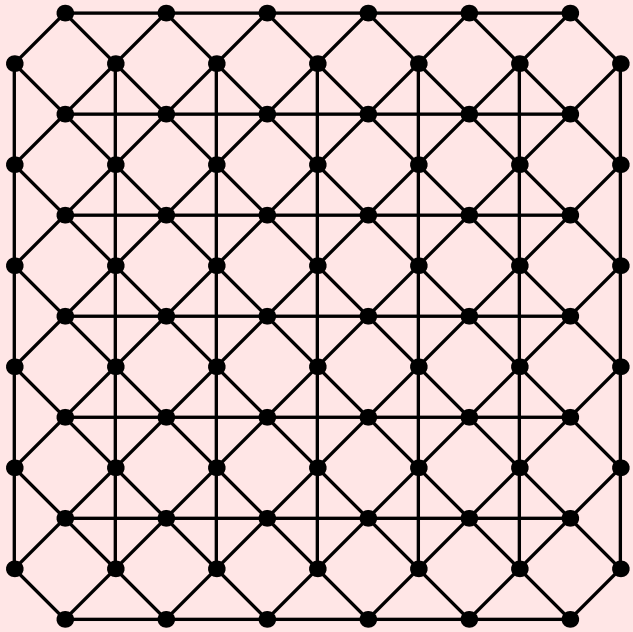
```
- Graphics -
```

```
ShowGraph[LineGraph[Cycle[10]]]
```



```
- Graphics -
```

```
ShowGraph[LineGraph[GridGraph[7, 7]]]
```



```
- Graphics -
```

```
a = DiscreteMath`OldCombinatorica`GridGraph[25, 25]; aa = GridGraph[25, 25];
```

```
{Timing[DiscreteMath`OldCombinatorica`LineGraph[a];], Timing[LineGraph[aa];]
```

```
{{19.656 Second, Null}, {0.219 Second, Null}}
```

NOTES

* The line graph inherits no graphics from the parent graph. It does inherit the embedding though.

* The speedup obtained with the new version is quite remarkable –19.672 seconds vs 0.297 seconds –in the above example.

TIMING DISCUSSION

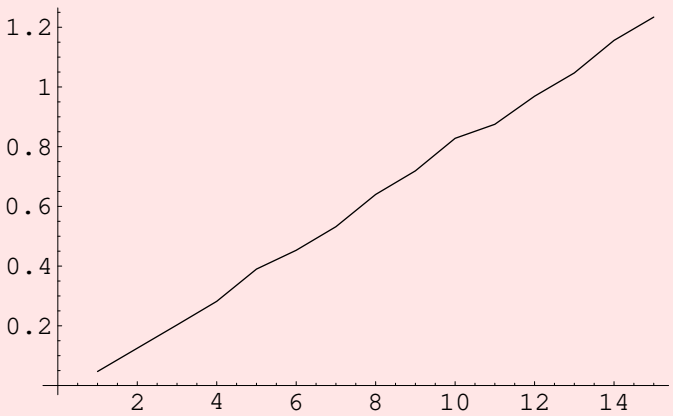
I need to look at the code carefully to analyze the asymptotic running time of the function. Ideally, this would be linear in the size of output. The line graph of a grid graph is linear is size as a function of the grid graph. So the following experiment should give us running times that grow in a linear fashion.

The plot below does seem to support this claim!

```
gt = Table[ GridGraph[20, 10 i], {i, 15}];
rt = Table[Timing[LineGraph[ gt[[i]]];], {i, 15}]
```

```
{{0.047 Second, Null}, {0.125 Second, Null}, {0.203 Second, Null},
 {0.282 Second, Null}, {0.39 Second, Null}, {0.453 Second, Null},
 {0.532 Second, Null}, {0.64 Second, Null}, {0.719 Second, Null},
 {0.828 Second, Null}, {0.875 Second, Null}, {0.969 Second, Null},
 {1.047 Second, Null}, {1.156 Second, Null}, {1.234 Second, Null}}
```

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



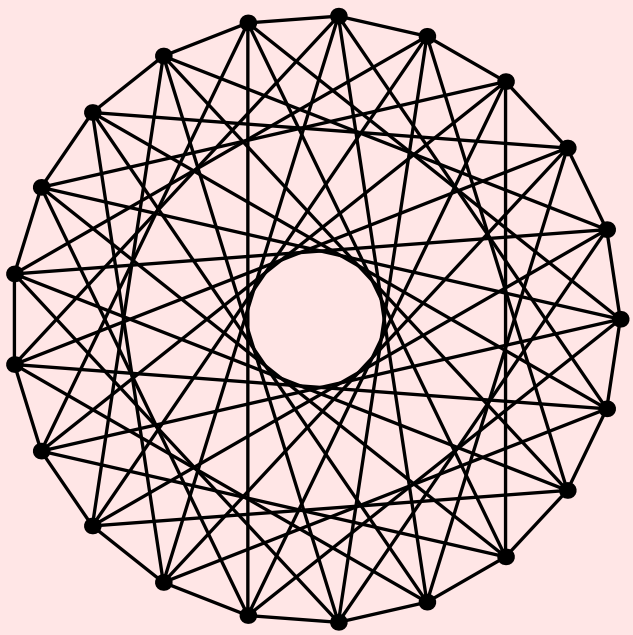

```
- Graphics -
```

CirculantGraph

```
?CirculantGraph
```

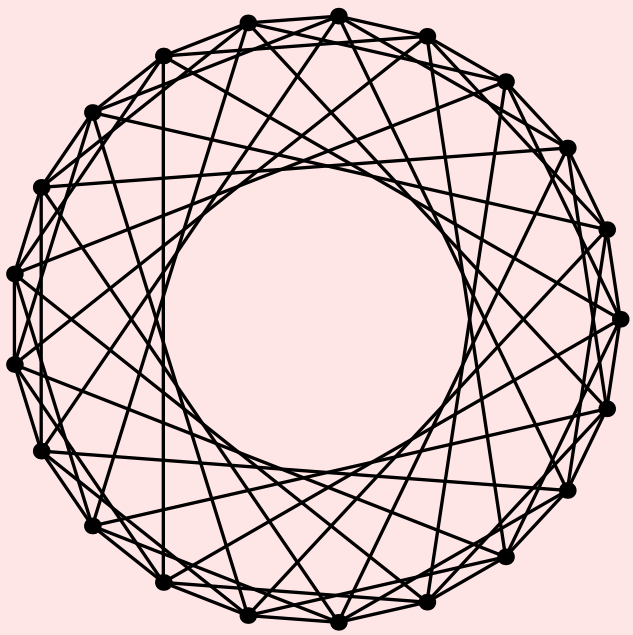
```
CirculantGraph[n, l] constructs a circulant graph on n vertices, meaning
  the ith vertex is adjacent to the (i+j)th and (i-j)th vertex, for each
  j in list l. CirculantGraph[n, l], where l is an integer, returns the
  graph with n vertices in which each i is adjacent to (i+l) and (i-l).
```

```
ShowGraph[CirculantGraph[21, RandomKSubset[Range[10], 3]]]
```



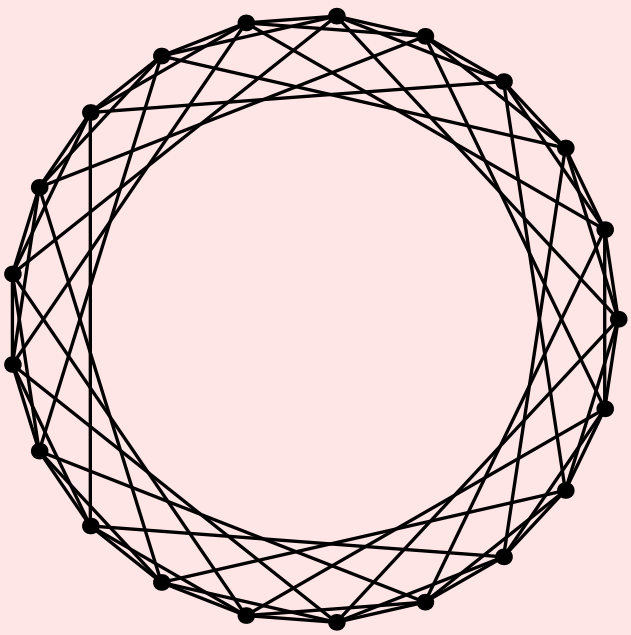
- Graphics -

```
ShowGraph[CirculantGraph[21, RandomKSubset[Range[10], 3]]]
```



```
- Graphics -
```

```
ShowGraph[CirculantGraph[21, RandomKSubset[Range[10], 3]]]
```



```
- Graphics -
```

NOTES

* Note that circulant graphs contain multiple edges. Here is an illustration. For odd n, using the subset Range[Floor[n/2]] generates a complete graph with each edge doubled. For even n, using the same subset generates a complete graph with each edge double plus edges of the form{i, i+n/2} for each i. This is a problem in the implementation of CompleteGraph – I could always use CirculantGraph and throw away the duplicates, but I am not sure I want to do that.

```
ne = Edges[CirculantGraph[5, {1, 2}]]
```

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

```
Length[ne]
```

```
10
```

```
ne = Edges[CirculantGraph[4, {1, 2}]]
```

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

```
Length[ne]
```

```
6
```

TIMING DISCUSSION

As expected, the running time of the CirculantGraph[...] function is linear in the size of the graph it produces. The function is quite fast and produces a circulant graph with 4000 vertices and 20,000 edges in about 3 seconds. However, the older implementation is faster because it simply uses a rotation operation on the adjacency matrix. But, using the older implementation we cannot come close to generating such large graphs.

```
{Timing[
  DiscreteMath`OldCombinatorica`CirculantGraph[2000, {1, 2, 3, 4, 5}];],
 Timing[CirculantGraph[2000, {1, 2, 3, 4, 5}];]}
```

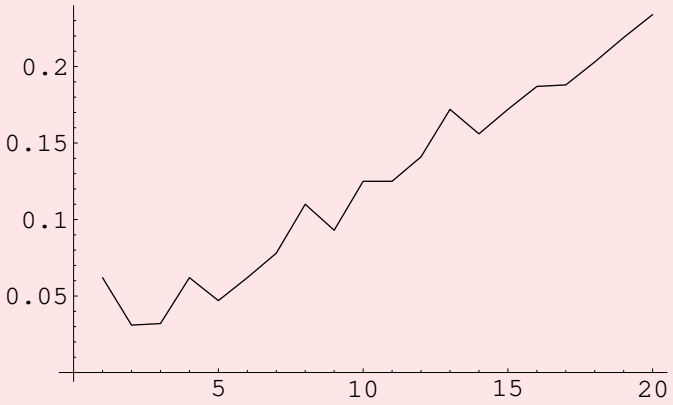
```
{{0.063 Second, Null}, {0.109 Second, Null}}
```

```
p = Table[ RandomKSubset[ Range[200 i] , 5], {i, 20}];
```

```
rt = Table[Timing[ CirculantGraph[200 i, p[[i]] ];], {i, 20}]
```

```
{{0.062 Second, Null}, {0.031 Second, Null},
 {0.032 Second, Null}, {0.062 Second, Null}, {0.047 Second, Null},
 {0.062 Second, Null}, {0.078 Second, Null}, {0.11 Second, Null},
 {0.093 Second, Null}, {0.125 Second, Null}, {0.125 Second, Null},
 {0.141 Second, Null}, {0.172 Second, Null}, {0.156 Second, Null},
 {0.172 Second, Null}, {0.187 Second, Null}, {0.188 Second, Null},
 {0.203 Second, Null}, {0.219 Second, Null}, {0.234 Second, Null}}
```

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



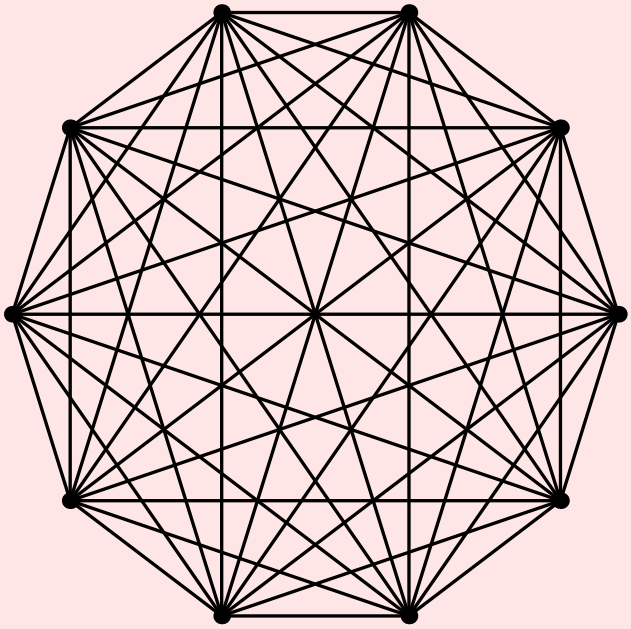

```
- Graphics -
```

CompleteGraph

```
?CompleteGraph
```

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

```
ShowGraph[CompleteGraph[10]]
```



```
- Graphics -
```

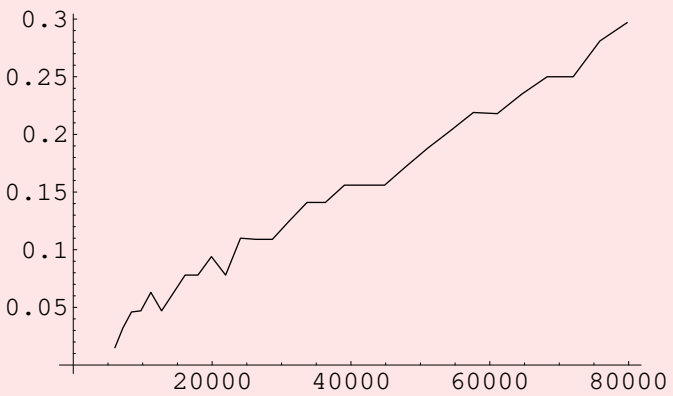
TIMING DISCUSSION

As expected, the running time of the CompleteGraph[...] is linear in the size of the graph. The function is pretty fast –a 400 vertex complete graph (with roughly, 80,000 edges) takes less than a second to be generated. The older implementation is faster since CirculantGraph[...] is faster, but that implementation cannot some close to generating graphs of the size that can be generated using the new implementation.

```
rt = Table[ Timing[ CompleteGraph[100 + 10 i];], {i, 30}]
```

```
{{0.015 Second, Null}, {0.032 Second, Null}, {0.046 Second, Null},
 {0.047 Second, Null}, {0.063 Second, Null}, {0.047 Second, Null},
 {0.062 Second, Null}, {0.078 Second, Null}, {0.078 Second, Null},
 {0.094 Second, Null}, {0.078 Second, Null}, {0.11 Second, Null},
 {0.109 Second, Null}, {0.109 Second, Null}, {0.125 Second, Null},
 {0.141 Second, Null}, {0.141 Second, Null}, {0.156 Second, Null},
 {0.156 Second, Null}, {0.156 Second, Null}, {0.172 Second, Null},
 {0.188 Second, Null}, {0.203 Second, Null}, {0.219 Second, Null},
 {0.218 Second, Null}, {0.235 Second, Null}, {0.25 Second, Null},
 {0.25 Second, Null}, {0.281 Second, Null}, {0.297 Second, Null}}
```

```
ListPlot[ Table[ { (50 + 5 i) (99 + 10 i), rt[[i, 1, 1]]}, {i, 30}],
 PlotJoined -> True]
```




```
- Graphics -
```

```
{Timing[DiscreteMath`OldCombinatorica`CompleteGraph[250];],
 Timing[CompleteGraph[250];]}
```

```
{{0.015 Second, Null}, {0.079 Second, Null}}
```

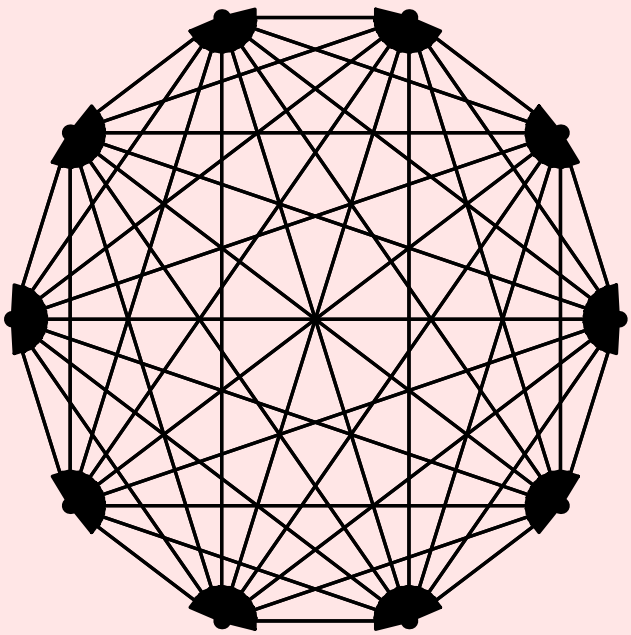
```
{Timing[DiscreteMath`OldCombinatorica`CompleteGraph[500];],
 Timing[CompleteGraph[500];]}
```

```
{{0.031 Second, Null}, {0.375 Second, Null}}
```

NOTES

* Complete directed graphs can also be generated.

```
ShowGraph[CompleteGraph[10, Type -> Directed]]
```

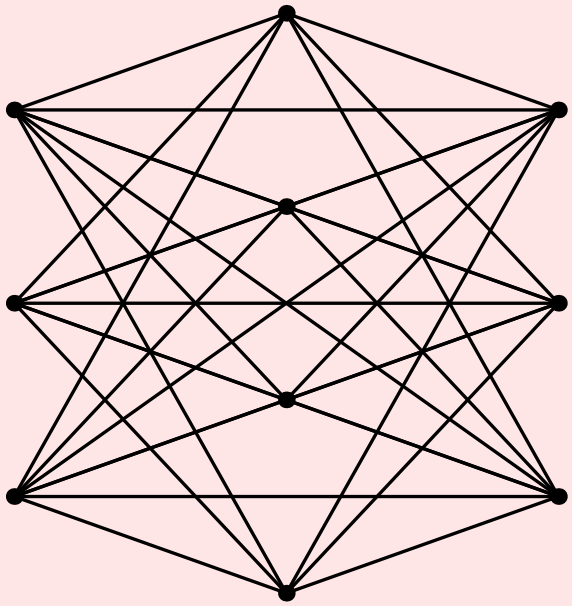


```
- Graphics -
```

TO DO

* Complete bipartite graphs can also be generated using CompleteGraph[....]. This seems to be too much of function name overloading. The graphs are different and the code is also quite different. I should just use something like CompleteKPartite Graph[...] instead.

```
ShowGraph[CompleteGraph[3, 4, 3]]
```

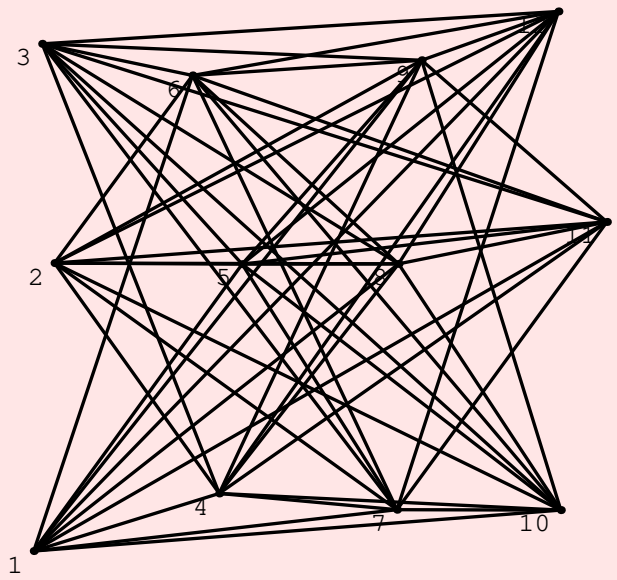


```
- Graphics -
```

```
? ShakeGraph
```

```
ShakeGraph[g, d] performs a random perturbation
  of the vertices of graph g, with each vertex moving,
  at most, a distance d from its original position.
```

```
ShowGraph[ShakeGraph[CompleteGraph[3, 3, 3, 3], 0.3],
 VertexStyle -> Disc[Small], VertexNumber -> On]
```




```
- Graphics -
```

```
Timing[ CompleteGraph[100, 100, 100, 100]; ]
```

```
{0.391 Second, Null}
```

```
Timing[ DiscreteMath`OldCombinatorica`CompleteGraph[100, 100, 100, 100];]
```

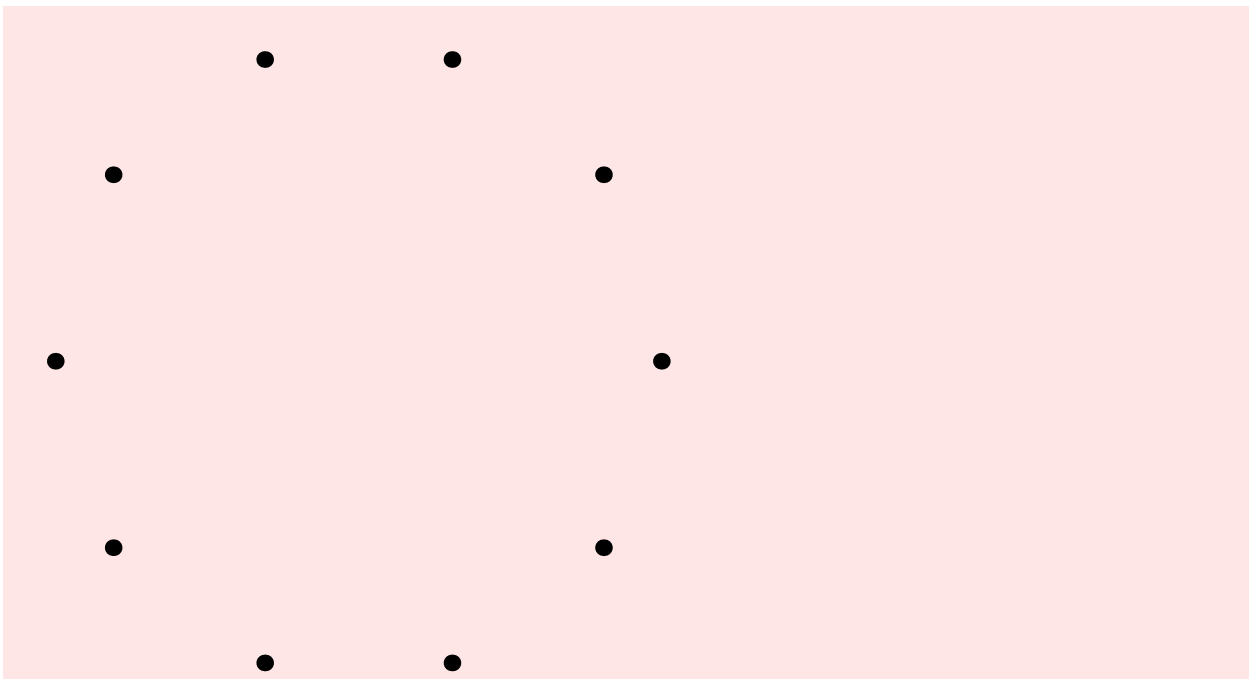
```
{0. Second, Null}
```

EmptyGraph

```
? EmptyGraph
```

```
EmptyGraph[n] generates an empty graph on n vertices. An
  option Type that can take on values Directed or Undirected
  is provided. The default setting is Type -> Undirected.
```

```
ShowGraph[EmptyGraph[10]]
```



```
- Graphics -
```

```
DiscreteMath`OldCombinatorica`ShowGraph[
 DiscreteMath`OldCombinatorica`EmptyGraph[10]]
```

```
- Graphics -
```

NOTES

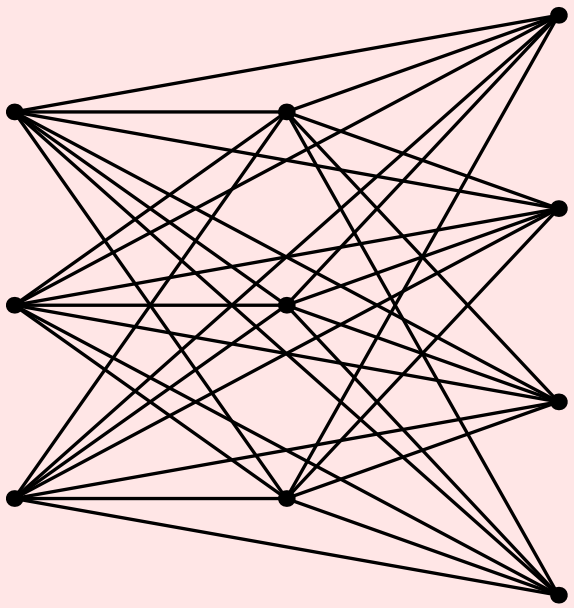
* Which embedding is better?

Turan

```
? Turan
```

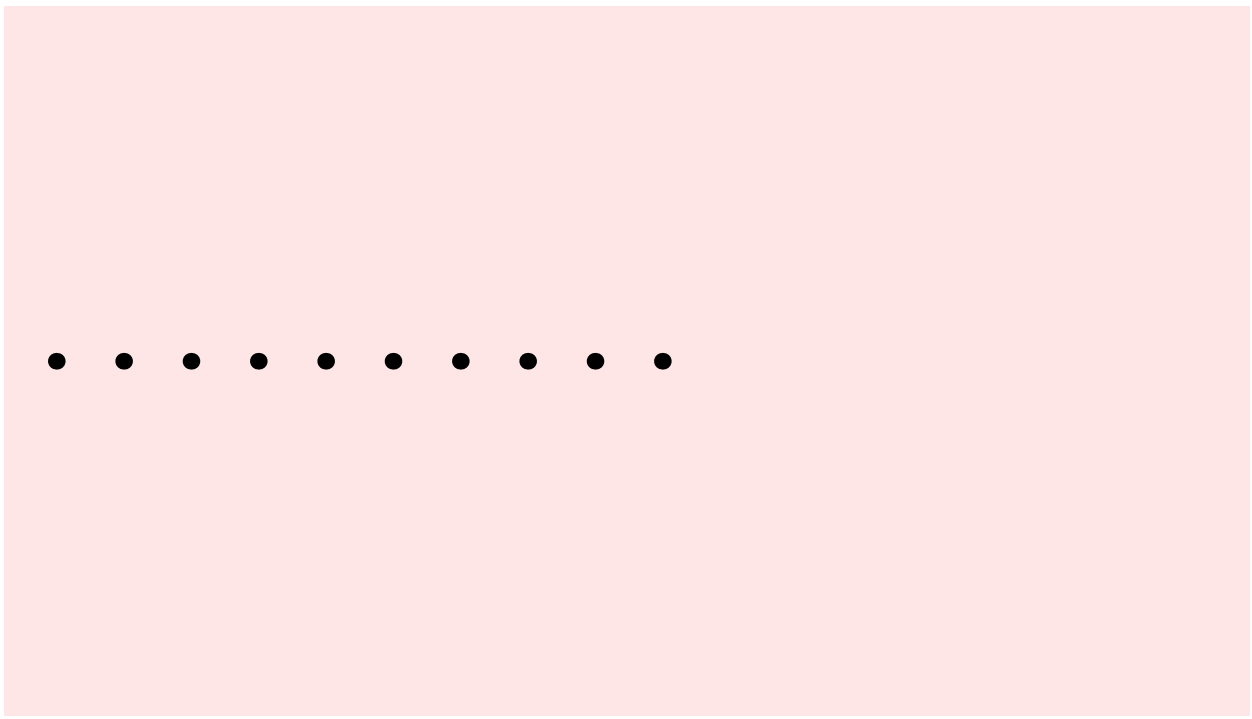
```
Turan[n, p] constructs the Turan graph, the extremal
  graph on n vertices that does not contain CompleteGraph[p].
```

```
ShowGraph[Turan[10, 4]]
```



```
- Graphics -
```

```
ShowGraph[Turan[10, 2]]
```



```
- Graphics -
```

TIMING DISCUSSION

The plot below confirms to some extent that Turan is linear in the size of the output it produces. I need to analyze the code further to make sure that this is true. The old implementation of Turan is much faster than the new implementation. Turan is definitely candidate for speedup. To speedup Turan I need to speed up the CompleteKpartiteGraph[...] implementation.

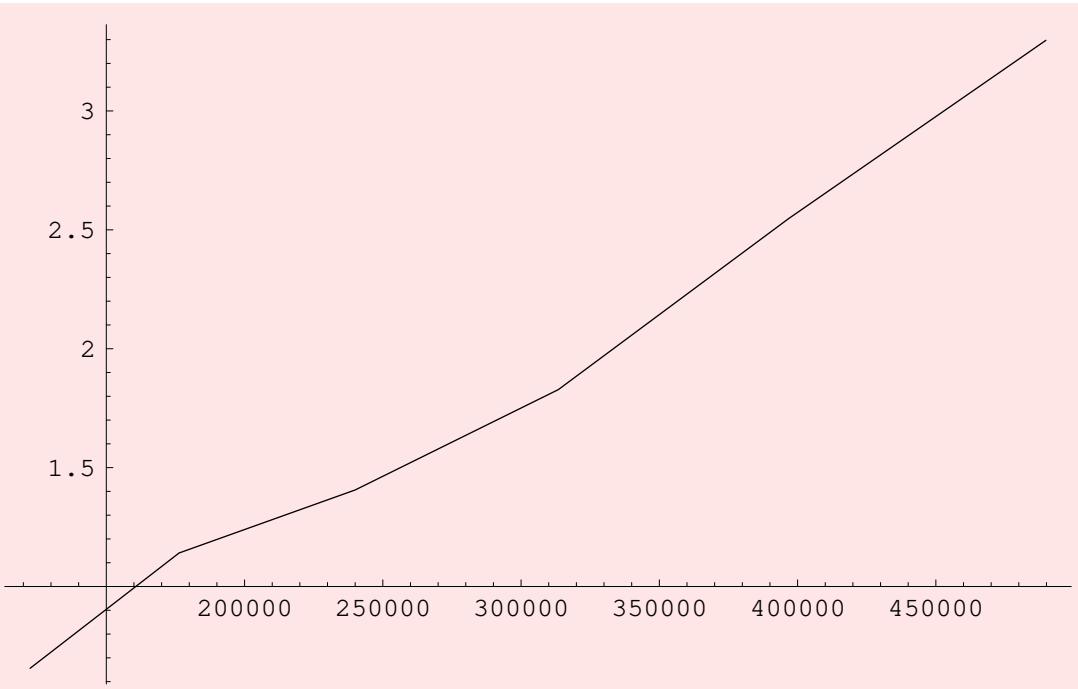
TO DO

Speedup the CompleteKpartiteGraph[...] implementation to make it more competitive with the older implementation.

```
rt = Table[Timing [gt[[i]] = Turan[i * 100, 50];], {i, 5, 10}]
```

```
{{0.656 Second, Null}, {1.141 Second, Null}, {1.406 Second, Null},
 {1.828 Second, Null}, {2.547 Second, Null}, {3.297 Second, Null}}
```

```
ListPlot[ Table[{M[ gt[[i]] ], rt[[ i - 4, 1, 1 ]] }, {i, 5, 10}],
 PlotJoined -> True, ImageSize -> 400]
```




```
- Graphics -
```

```
Table[Timing [ DiscreteMath`OldCombinatorica`Turan[i * 100, 50];], {i, 5, 10}]
```

```
{{0.015 Second, Null}, {0.047 Second, Null}, {0.047 Second, Null},
 {0.047 Second, Null}, {0.047 Second, Null}, {0.047 Second, Null}}
```

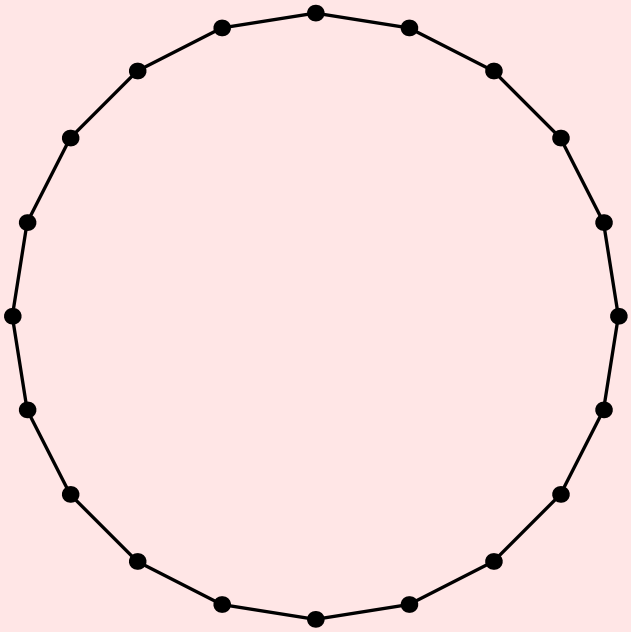
```
Clear[gt]
```

Cycle

```
?Cycle
```

```
Cycle[n] constructs the cycle on n vertices, a 2-regular
  connected graph. An option Type that takes on values Directed or
  Undirected is allowed. The default setting is Type -> Undirected.
```

```
ShowGraph[Cycle[20]]
```



```
- Graphics -
```

TIMING DISCUSSION

The old implementation of Cycle is quite fast given that it is working on the adjacency matrix. It is also surprising that my machine did not crash while attempting to construct the 2000 x 2000 adjacency matrix. Anyway, this is probably the limit for the old implementation. The new implementation manages to build a 50,000 vertex cyle in about 3 seconds!

Since the code of Cycle is rather simple, it is easy to see that its running time is θ(n). The plot below conforms this.

```
{Timing[DiscreteMath`OldCombinatorica`Cycle[2000];], Timing[Cycle[2000];]}
```

```
{{0.047 Second, Null}, {0.031 Second, Null}}
```

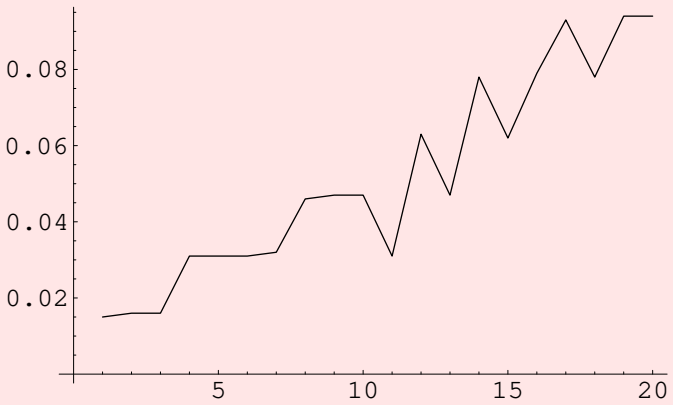
```
Timing[ Cycle[ 50000]; ]
```

```
{0.422 Second, Null}
```

```
rt = Table[ Timing[Cycle[500 i];] , {i, 20}]
```

```
{{0.015 Second, Null}, {0.016 Second, Null},
 {0.016 Second, Null}, {0.031 Second, Null}, {0.031 Second, Null},
 {0.031 Second, Null}, {0.032 Second, Null}, {0.046 Second, Null},
 {0.047 Second, Null}, {0.047 Second, Null}, {0.031 Second, Null},
 {0.063 Second, Null}, {0.047 Second, Null}, {0.078 Second, Null},
 {0.062 Second, Null}, {0.079 Second, Null}, {0.093 Second, Null},
 {0.078 Second, Null}, {0.094 Second, Null}, {0.094 Second, Null}}
```

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



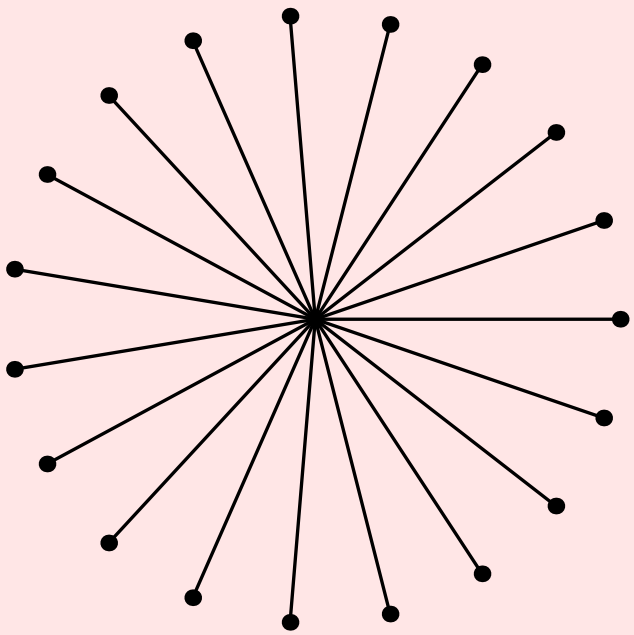

```
- Graphics -
```

Star

```
?Star
```

```
Star[n] constructs a star on n vertices,
  which is a tree with one vertex of degree n-1.
```

```
ShowGraph[Star[20]]
```



```
- Graphics -
```

TIMING DISCUSSION

Note that the old implementation of Star is quite slow as compared to the new implementation Star. As expected, the running time of Star is linear.

```
{Timing[DiscreteMath`OldCombinatorica`Star[1000];], Timing[Star[1000];]}
```

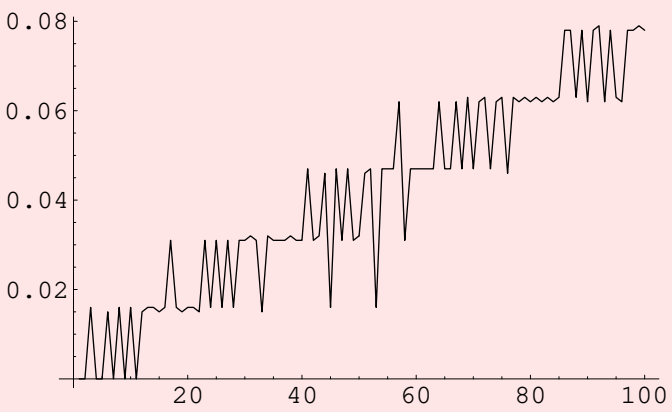
```
{{0.25 Second, Null}, {0. Second, Null}}
```

```
{Timing[DiscreteMath`OldCombinatorica`Star[2000];], Timing[Star[2000];]}
```

```
{{1.297 Second, Null}, {0.016 Second, Null}}
```

```
rt = Table[ Timing[ Star[100 i];], {i, 100}];
```

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



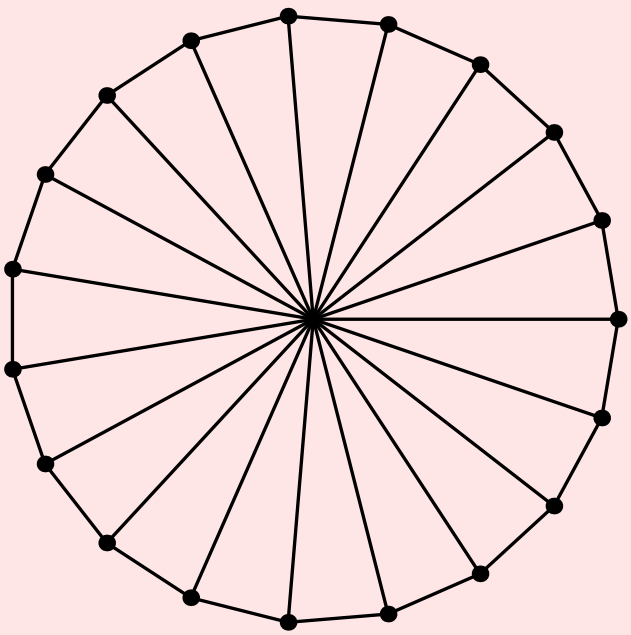

```
- Graphics -
```

Wheel

```
?Wheel
```

```
Wheel[n] constructs a wheel on n vertices,
  which is the join of CompleteGraph[1] and Cycle[n-1].
```

```
ShowGraph[Wheel[20]]
```



```
- Graphics -
```

TIMING DISCUSSION

The new implementation of Wheel is much faster than the old. As expected, the new implementation is linear.

```
{Timing[DiscreteMath`OldCombinatorica`Wheel[1000];], Timing[Wheel[1000];]}
```

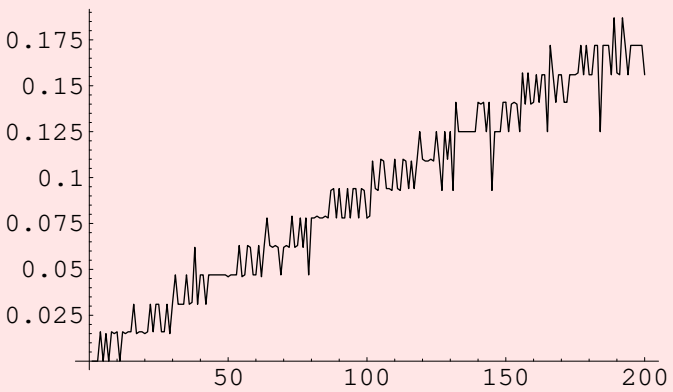
```
{{0.047 Second, Null}, {0.016 Second, Null}}
```

```
{Timing[DiscreteMath`OldCombinatorica`Wheel[2000];], Timing[Wheel[2000];]}
```

```
{{0.078 Second, Null}, {0.015 Second, Null}}
```

```
rt = Table[ Timing[ Wheel[100 i];], {i, 200}];
```

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

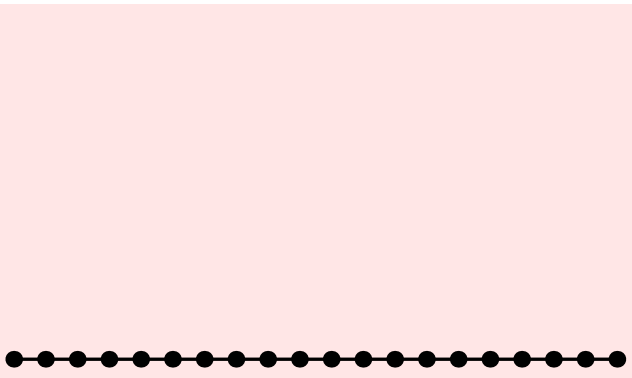



```
- Graphics -
```

Path

```
?Path
```

```
ShowGraph[Path[20]]
```



```
- Graphics -
```

TIMING DISCUSSION

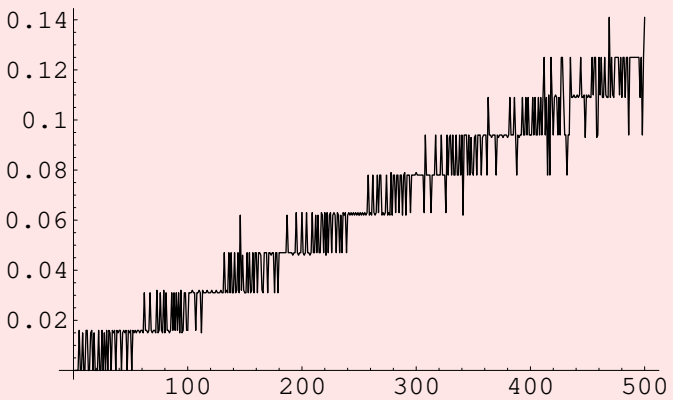
The old implementation of Path is ridiculously slow! The new implementation is extremely fast. A path of length 5000 can be constructed in 0.13 seconds. Since the running time of the new implementation is so small the plot is somewhat bumpy; it is linear nevertheless.

```
Timing[DiscreteMath`OldCombinatorica`Path[2000];]
```

```
{20.859 Second, Null}
```

```
rt = Table[ Timing[ Path[100 i]; ], {i, 500}];
```

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



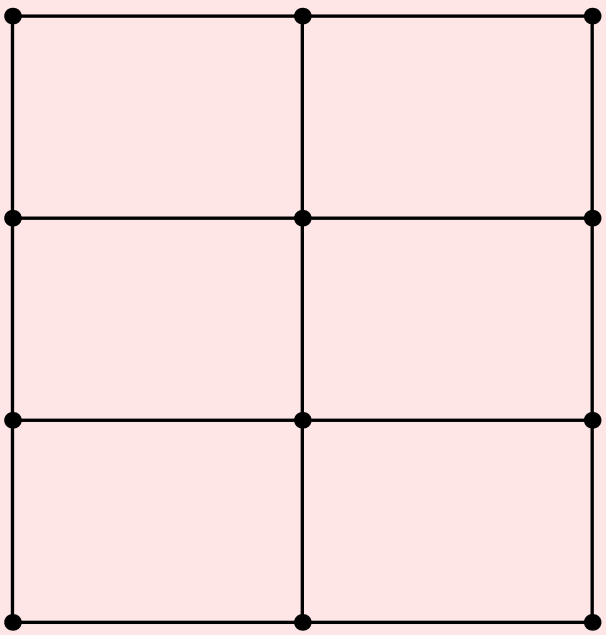

```
- Graphics -
```

GridGraph

```
?GridGraph
```

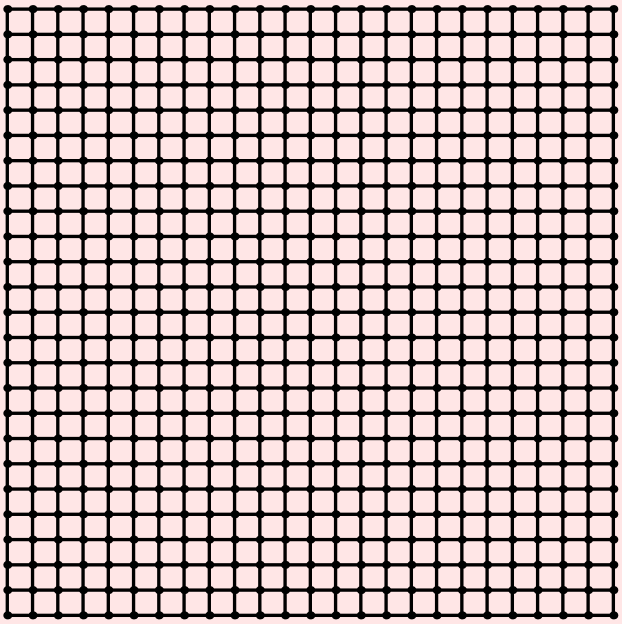
```
GridGraph[n, m] constructs an n*m grid graph, the product of paths on
  n and m vertices. GridGraph[p, q, r] constructs a p*q*r grid
  graph, the product of GridGraph[p, q] and a path of length r.
```

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



```
- Graphics -
```

```
ShowGraph[GridGraph[25, 25], VertexStyle -> Disc[Small]]
```



```
- Graphics -
```

TIMING DISCUSSION

The new implementation of GridGraph[...] is much faster than the old implementation. As expected, the running time of GridGraph is linear.

```
Timing[DiscreteMath`OldCombinatorica`GridGraph[40, 40];]
```

```
{9.109 Second, Null}
```

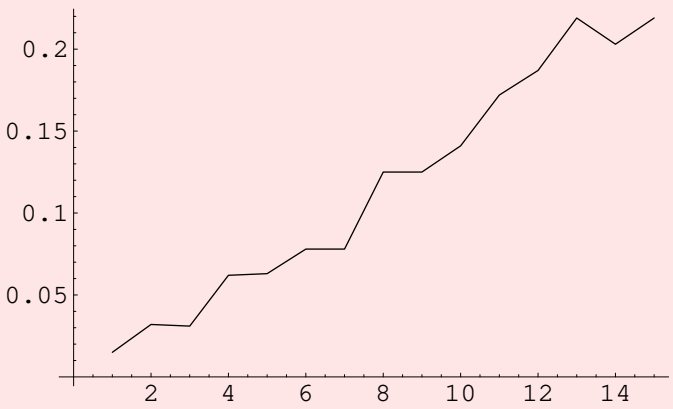
```
Timing[DiscreteMath`OldCombinatorica`GridGraph[30, 30];]
```

```
{2.813 Second, Null}
```

```
p = Table[Timing[GridGraph[20, 10 i];], {i, 15}]
```

```
{{0.015 Second, Null}, {0.032 Second, Null}, {0.031 Second, Null},
 {0.062 Second, Null}, {0.063 Second, Null}, {0.078 Second, Null},
 {0.078 Second, Null}, {0.125 Second, Null}, {0.125 Second, Null},
 {0.141 Second, Null}, {0.172 Second, Null}, {0.187 Second, Null},
 {0.219 Second, Null}, {0.203 Second, Null}, {0.219 Second, Null}}
```

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



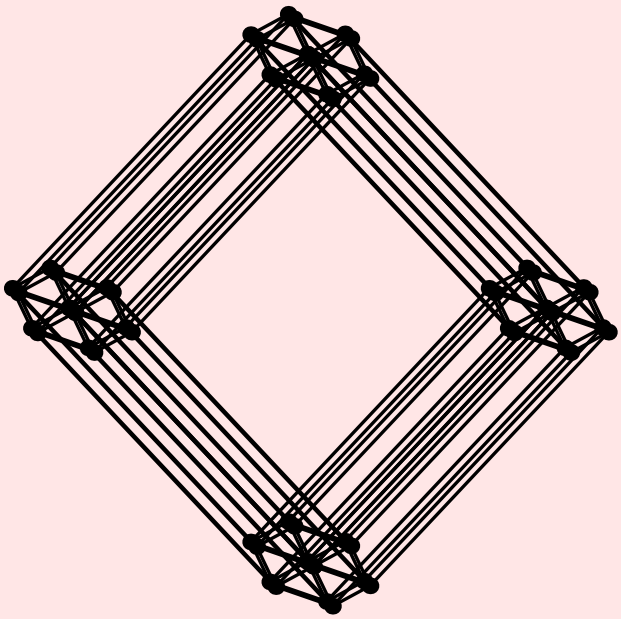

```
- Graphics -
```

HyperCube

```
?Hypercube
```

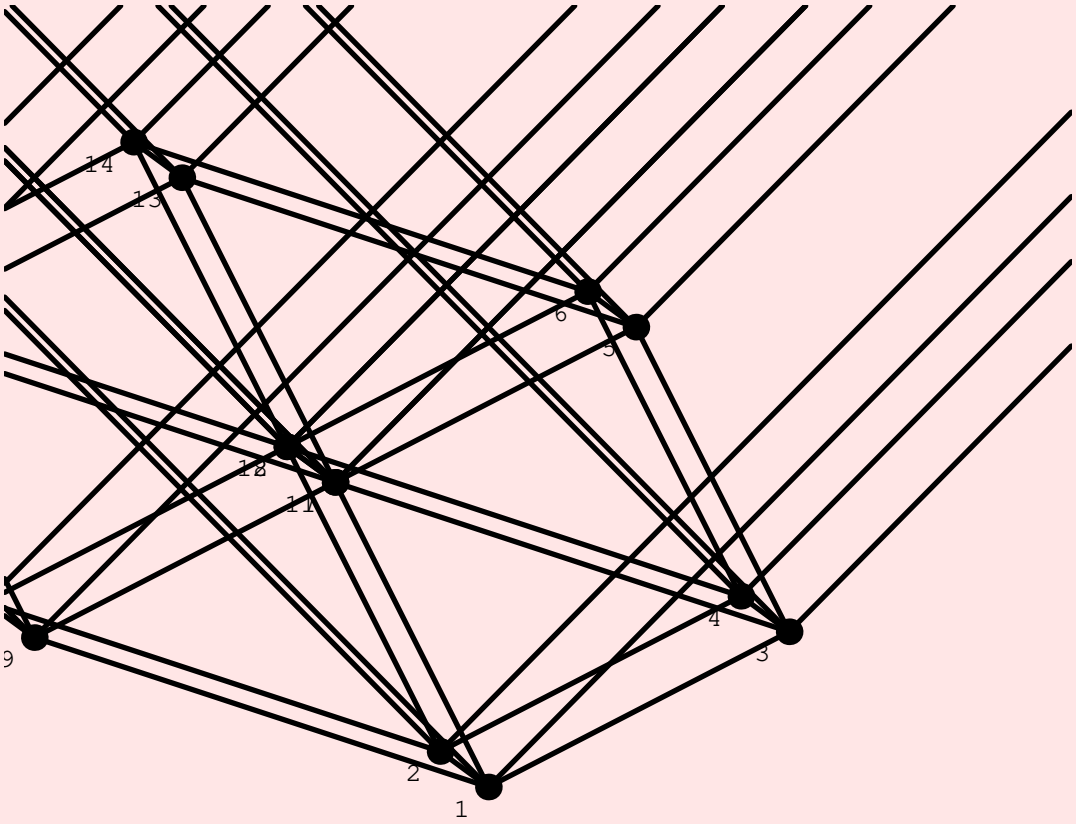
```
Hypercube[n] constructs an n-dimensional hypercube.
```

```
ShowGraph[Hypercube[6]]
```



```
- Graphics -
```

```
ShowGraph[Hypercube[6], PlotRange -> Zoom[{1, 2, 3, 4, 5, 6, 7, 8}],
 VertexNumber -> On, ImageSize -> 400]
```




```
- Graphics -
```

TIMING DISCUSSION

The new implementation of Hypercube is much faster than the old implementation. The implementations use dynamic programming to construct the hypercubes. Because of this the Table of timings below for the new implementation of Hypercube is misleading. Information stored while computing smaller hypercubes gets used in computing larger hypercubes. It is also somewhat peculiar that I was able to compute upto dimension 15 in the table, but ran out of memory when doing this separately.

```
Timing[DiscreteMath`OldCombinatorica`Hypercube[10];]
```

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

```
Timing[Hypercube[10];]
```

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

```
Table[Timing[Hypercube[n];], {n, 15}]
```

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

```
Hypercube[15]
```

```
-Graph:<245760, 32768, Undirected>-
```

LabeledTreeToCode

```
?LabeledTreeToCode
```

```
LabeledTreeToCode[g] reduces the tree g to its Prufer code.
```

```
LabeledTreeToCode[RandomTree[20]]
```

```
{14, 10, 11, 11, 4, 3, 17, 16, 20, 7, 18, 13, 4, 10, 9, 20, 4, 5}
```

```
LabeledTreeToCode[CodeToLabeledTree[Table[10, {20}]]]
```

```
{10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10}
```

```
LabeledTreeToCode[Path[100]]
```

```
{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}
```

TIMING DISCUSSION

The implementation of this function seems to be linear based on the plot below. However, I need to look at code some more to make sure that this is indeed true. However, this function is a candidate for speedup since it takes about 5 s to construct a Prufer code with 1000 vertices.

The speedup in going from the old implementation to the new implementation seems to be roughly a factor of 6.

TO DO

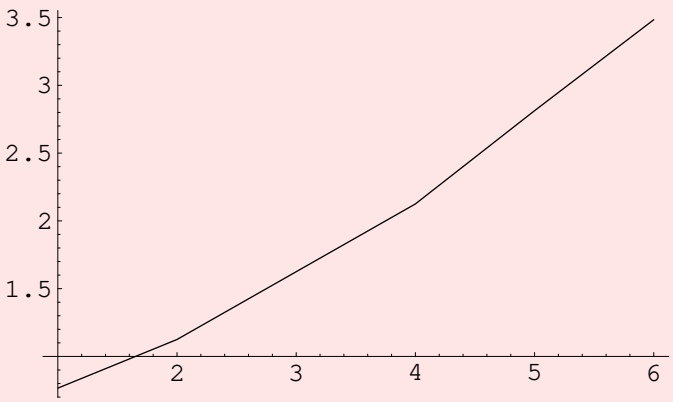
Speedup LabeledTreeToCode[...]

```
g = Table[RandomTree[i], {i, 500, 1000, 100}];
```

```
rt = Table[Timing[LabeledTreeToCode[ g[[i]] ]; ], {i, 6}]
```

```
{{0.766 Second, Null}, {1.125 Second, Null}, {1.625 Second, Null},
 {2.125 Second, Null}, {2.813 Second, Null}, {3.484 Second, Null}}
```

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




```
- Graphics -
```

```
gg = Table[DiscreteMath`OldCombinatorica`RandomTree[i], {i, 500, 1000, 500}]
```

```
Table[
 Timing[ DiscreteMath`OldCombinatorica`LabeledTreeToCode[ gg[[i]] ]; ], {i, 2}]
```

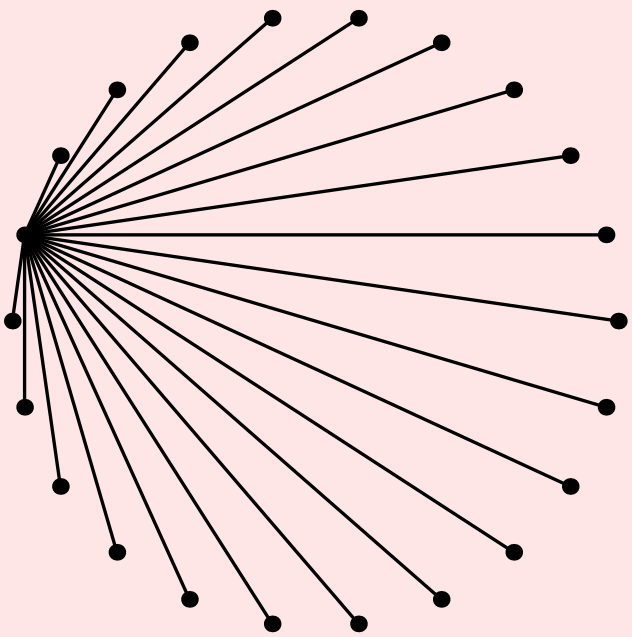
```
{{2.234 Second, Null}, {8.86 Second, Null}}
```

CodeToLabeledTree

```
?CodeToLabeledTree
```

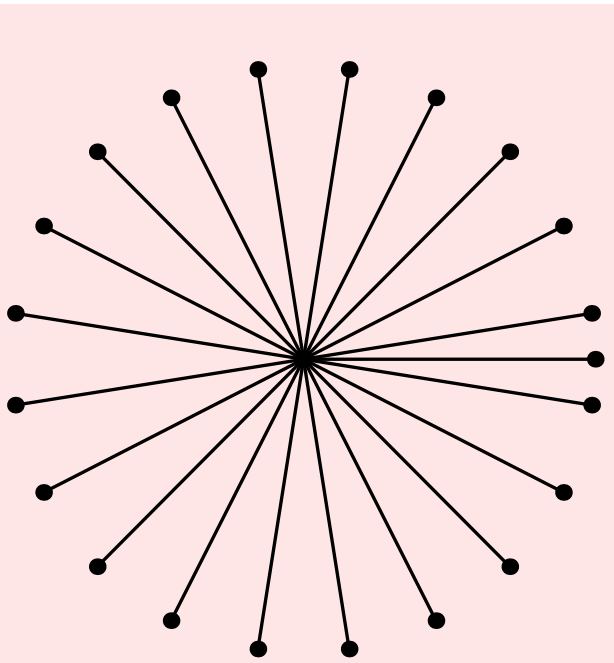
```
CodeToLabeledTree[l] constructs the unique
  labeled tree on n vertices from the Prufer code l,
  which consists of a list of n-2 integers between 1 and n.
```

```
ShowGraph[CodeToLabeledTree[Table[10, {20}]]]
```



```
- Graphics -
```

```
ShowGraph[RadialEmbedding[CodeToLabeledTree[Table[10, {20}]], 1]]
```



```
- Graphics -
```

TIMING DISCUSSION

A quick look at the code of CodeToLabeledTree reveals that the asymptotic running time of this function is $\theta(n^2 \log(n))$. This is shown to some extent in the plot below −though it is obviously difficult to conclude the log(n) factor. This function is definitely a candidate for speedup. The new implementation is about 5 times faster than the old implementation.

TO DO

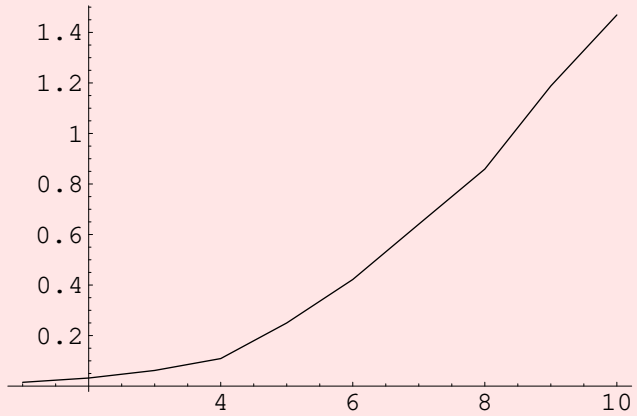
Speed up the CodeToLabeledTree function

```
s = Table[Table[Random[Integer, {1, i * 100}] , {i * 100 - 2}], {i, 10}];
```

```
rt = Table[ Timing[ CodeToLabeledTree[s[[i]] ]; ], {i, 10}]
```

```
{{0.015 Second, Null}, {0.032 Second, Null},
 {0.062 Second, Null}, {0.109 Second, Null},
 {0.25 Second, Null}, {0.422 Second, Null}, {0.641 Second, Null},
 {0.859 Second, Null}, {1.188 Second, Null}, {1.469 Second, Null}}
```

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




```
- Graphics -
```

```
Table[
 Timing[ DiscreteMath`OldCombinatorica`CodeToLabeledTree [s[[i]] ]; ], {i, 3}]
```

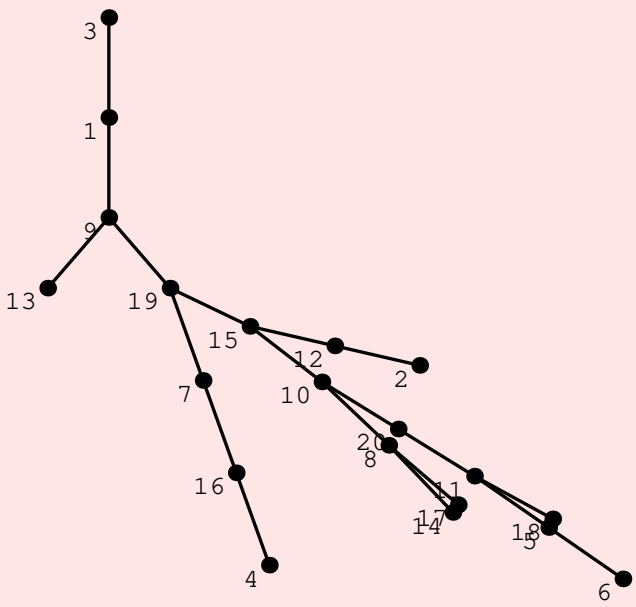
```
{{0.062 Second, Null}, {0.344 Second, Null}, {0.484 Second, Null}}
```

RandomTree

```
? RandomTree
```

```
RandomTree[n] constructs a random labeled tree on n vertices.
```

```
ShowGraph[RadialEmbedding[RandomTree[20], 1], VertexNumber -> On]
```




```
- Graphics -
```

TIMING DISCUSSION

The new version of RandomTree seems about 4–5times as fast as the old version. However, the new version might be speeded up if CodeToLabeledTree is speeded up. That might be possible by somehow getting rid of the Complement operations.

RandomTree generates a random code and then uses CodeToLabeledTree to get a tree. So its running time is completely dominated by the running time of CodeToLabeledTree. So it is not surprising that the running time plot looks quadratic.

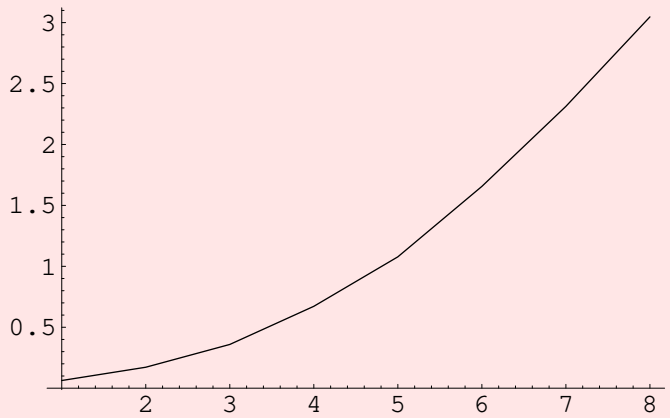
```
Table[Timing[DiscreteMath`OldCombinatorica`RandomTree[i];],
 {i, 100, 800, 100}]
```

```
{{0.203 Second, Null}, {0.672 Second, Null},
 {1.234 Second, Null}, {2.25 Second, Null}, {4.063 Second, Null},
 {5.796 Second, Null}, {7.797 Second, Null}, {9.922 Second, Null}}
```

```
p = Table[Timing[RandomTree[i];], {i, 100, 800, 100}]
```

```
{{0.063 Second, Null}, {0.172 Second, Null},
 {0.359 Second, Null}, {0.672 Second, Null}, {1.078 Second, Null},
 {1.656 Second, Null}, {2.313 Second, Null}, {3.047 Second, Null}}
```

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



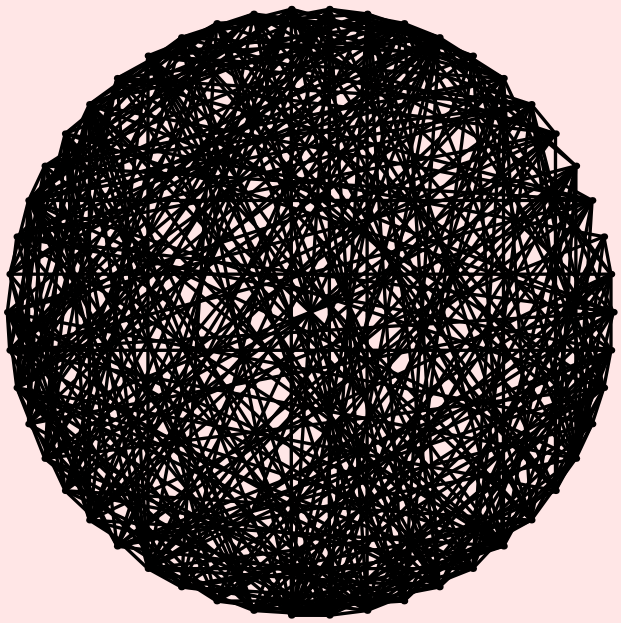

```
- Graphics -
```

RandomGraph

```
? RandomGraph
```

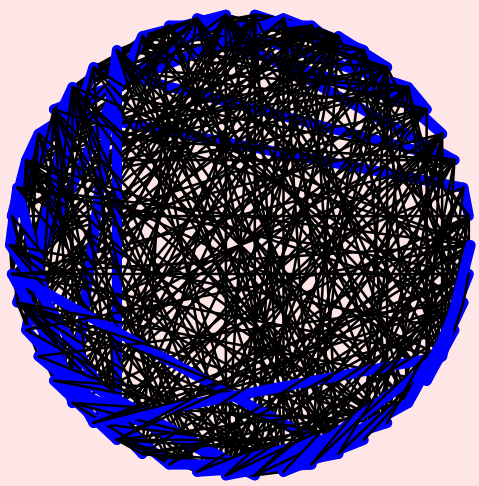
```
RandomGraph[n, p] constructs a random labeled graph on n vertices with
  an edge probability of p. An option Type is provided, which
  can take on values Directed and Undirected, and whose default
  value is Undirected. Type->Directed produces a corresponding
  random directed graph. The usages Random[n, p, Directed],
  Random[n, p, range] and Random[n, p, range, Directed] are
  all obsolete. Use SetEdgeWeights to set random edge weights.
```

```
ShowGraph[s = RandomGraph[50, .3], VertexStyle -> Disc[0.01]]
```



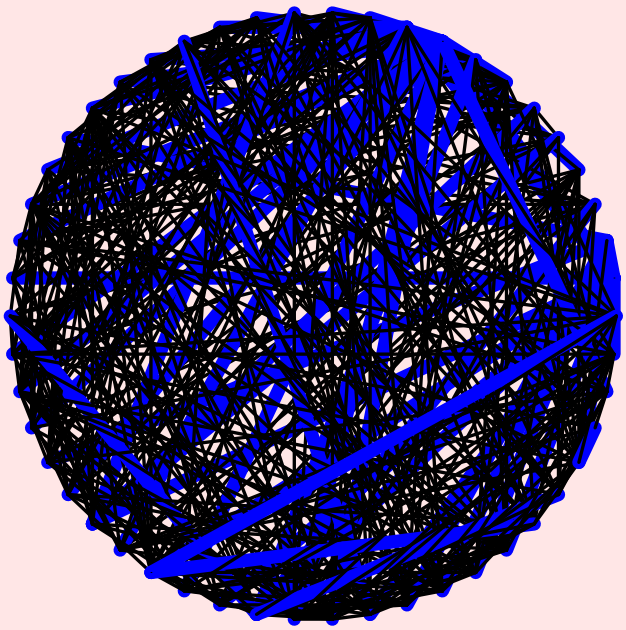
```
- Graphics -
```

```
ShowGraph[Highlight[s, {Map[Sort, DepthFirstTraversal[s, 1, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexStyle -> Disc[0]]
```



```
- Graphics -
```

```
ShowGraph[Highlight[s, {Map[Sort, BreadthFirstTraversal[s, 1, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexStyle -> Disc[0]]
```



```
- Graphics -
```

TIMING DISCUSSION

RandomGraph[..] takes time $\theta(n^2 + m)$, where n and m are the number of vertices and edges in the produced random graph. This is supported by the plots below. In the first plot n is fixed while p increases in linear fashion. In the second plot n increases and p is set at 1/n. This means that the expected number of edges is linear in n. However, the running time in this case increases in a quadratic fashion. It would wonderful if I could implement RandomGraph[...] so that its running time is θ(m+n).

TO DO

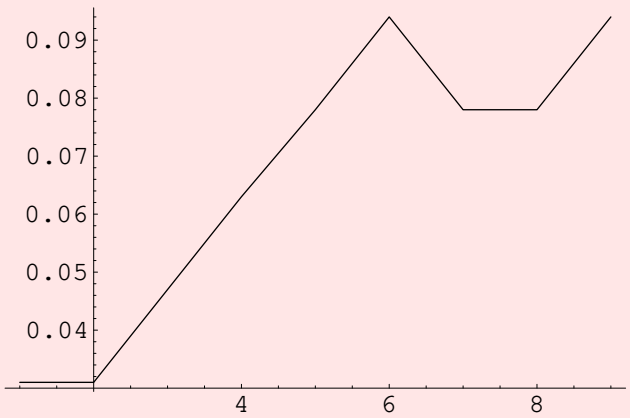
Implement RandomGraph[...] so that it takes time θ(m+n).

The new RandomGraph is much slower than the old RandomGraph because the new RandomGraph first constructs a random adjacency matrix (like RandomGraph) and then converts it into an edge list representation of a graph.

```
rt = Table[ Timing[ RandomGraph[100, p];], {p, 0.1, 0.9, 0.1}]
```

```
{{0.031 Second, Null}, {0.031 Second, Null}, {0.047 Second, Null},
 {0.063 Second, Null}, {0.078 Second, Null}, {0.094 Second, Null},
 {0.078 Second, Null}, {0.078 Second, Null}, {0.094 Second, Null}}
```

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

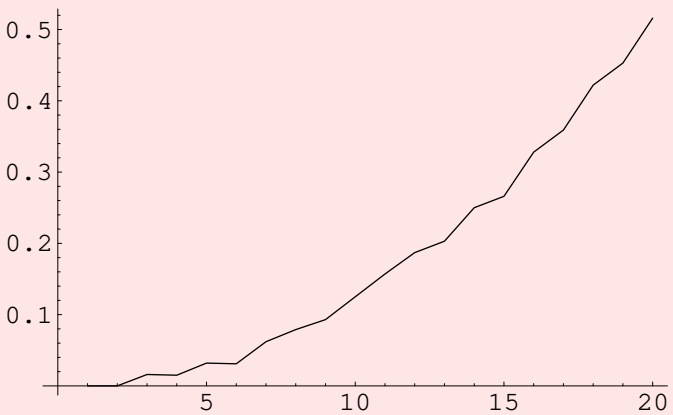



```
- Graphics -
```

```
rt = Table[ Timing[ RandomGraph[20 i, N[1 / (20 i)]];], {i, 20}]
```

```
{{0. Second, Null}, {0. Second, Null},
 {0.016 Second, Null}, {0.015 Second, Null}, {0.032 Second, Null},
 {0.031 Second, Null}, {0.062 Second, Null}, {0.079 Second, Null},
 {0.093 Second, Null}, {0.125 Second, Null}, {0.157 Second, Null},
 {0.187 Second, Null}, {0.203 Second, Null}, {0.25 Second, Null},
 {0.266 Second, Null}, {0.328 Second, Null}, {0.359 Second, Null},
 {0.422 Second, Null}, {0.453 Second, Null}, {0.516 Second, Null}}
```

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




```
- Graphics -
```

```
{Timing[DiscreteMath`OldCombinatorica`RandomGraph[100, .5];],
 Timing[RandomGraph[100, .5];]}
```

```
{{0.031 Second, Null}, {0.063 Second, Null}}
```

```
{Timing[DiscreteMath`OldCombinatorica`RandomGraph[300, .5];],
 Timing[RandomGraph[300, .5];]}
```

```
{{0.281 Second, Null}, {0.516 Second, Null}}
```

```
{Timing[ DiscreteMath`OldCombinatorica`RandomGraph[300, .8];],
 Timing[RandomGraph[300, .8];]}
```

```
{{0.312 Second, Null}, {0.703 Second, Null}}
```

BUG

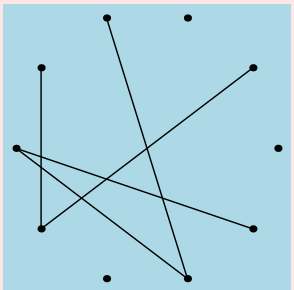
Another problem with the new RandomGraph is that a version of the form RandomGraph[n, p, range] has been provided though, this does not work as of now. I have to decide whether to make this work or give separate options for generating random multigraphs and non−simplegraphs.

ExactRandomGraph

```
?ExactRandomGraph
```

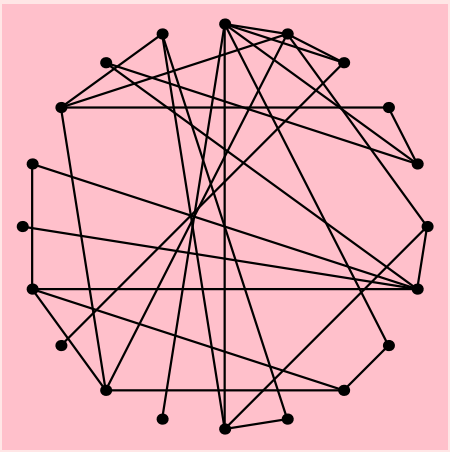
```
ExactRandomGraph[n, e] constructs a random
  labeled graph of exactly e edges and n vertices.
```

```
ShowGraph[ExactRandomGraph[10, 5], Background -> LightBlue]
```



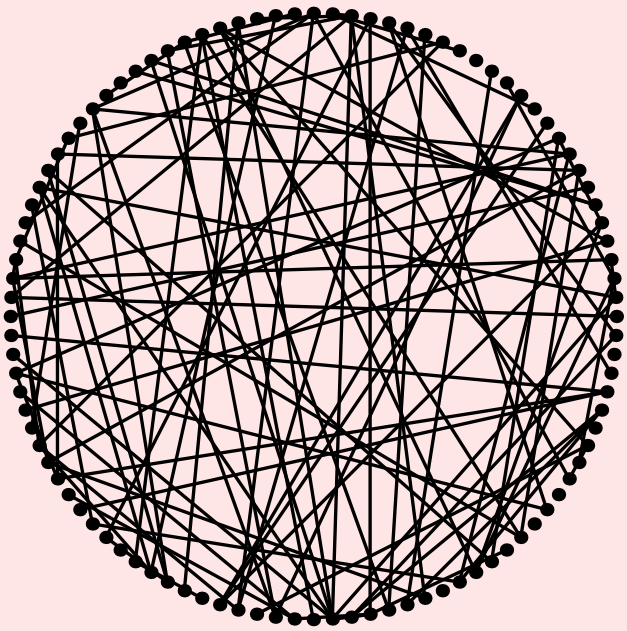
```
- Graphics -
```

```
ShowGraph[ExactRandomGraph[20, 30], Background -> Pink]
```



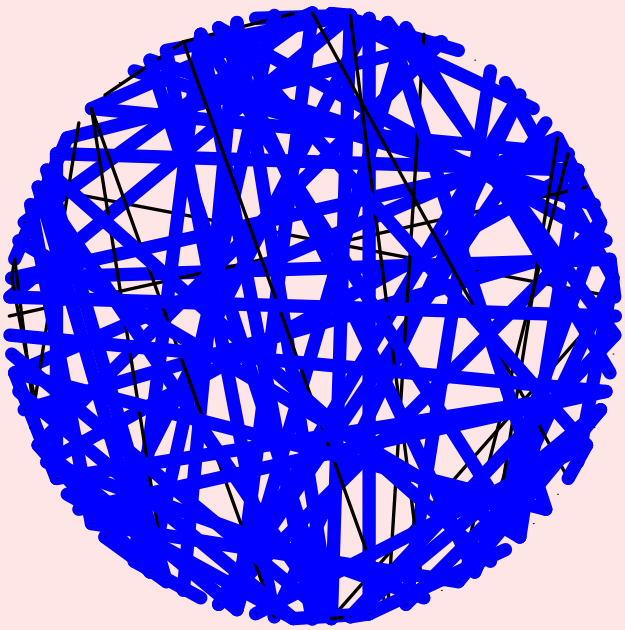
```
- Graphics -
```

```
ShowGraph[s = ExactRandomGraph[100, 100], VertexStyle -> Disc[0.02]]
```



```
- Graphics -
```

```
ShowGraph[Highlight[s, {Map[Sort, DepthFirstTraversal[s, 1, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexStyle -> Disc[0]]
```



```
- Graphics -
```

TIMING DISCUSSON

This function has running time $\theta(mn+n^2)$ because it first generates a random permutation of size n(n−1)/2,picks the first m elements and then unranks these numbers into a set of edges. Each Unrank operation (using the function NthPair) takes $\theta(n)$ time. This is unfortunate and I wonder if this can be done in $\theta(m + n)$ time. The following plot shows the linear dependence on m and the plot below that shows the quadratic dependence on n.

The new ExactRandomGraph is not much faster than the old ExactRandomGraph. The small difference comes about because the new ExactRandomGraph does not need to convert from edges to adjacency matrix.

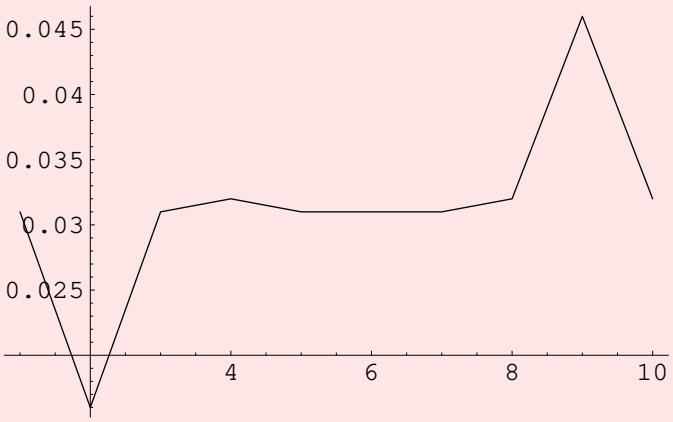
TO DO

Speed up ExactRandomGraph[...] so that it takes $\theta(m + n)$ time.

```
rt = Table[ Timing[ExactRandomGraph[100, m];], {m, 100, 1000, 100}]
```

```
{{0.031 Second, Null}, {0.016 Second, Null},
 {0.031 Second, Null}, {0.032 Second, Null},
 {0.031 Second, Null}, {0.031 Second, Null}, {0.031 Second, Null},
 {0.032 Second, Null}, {0.046 Second, Null}, {0.032 Second, Null}}
```

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

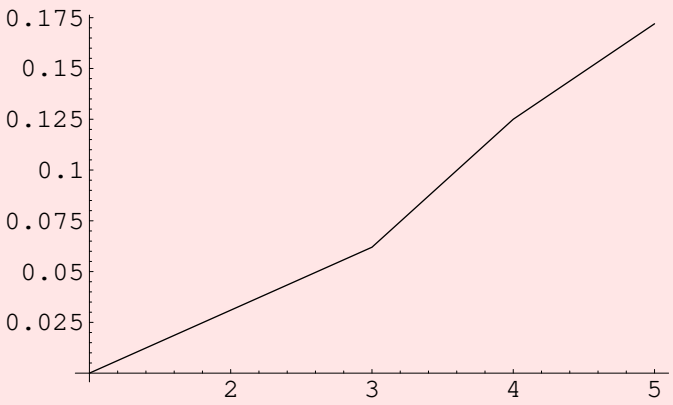



```
- Graphics -
```

```
rt = Table[ Timing[ExactRandomGraph[n, 5 n];], {n, 50, 250, 50}]
```

```
{{0. Second, Null}, {0.031 Second, Null},
 {0.062 Second, Null}, {0.125 Second, Null}, {0.172 Second, Null}}
```

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




```
- Graphics -
```

```
{Timing[DiscreteMath`OldCombinatorica`ExactRandomGraph[100, 100];],
 Timing[ExactRandomGraph[100, 100];]}
```

```
{{0.219 Second, Null}, {0.031 Second, Null}}
```

```
{Timing[DiscreteMath`OldCombinatorica`ExactRandomGraph[500, 500];],
 Timing[ExactRandomGraph[500, 500];]}
```

```
{{4.782 Second, Null}, {0.75 Second, Null}}
```

```
Timing[Map[{NthPair[#]} &, Take[RandomPermutation[500 (500 - 1) / 2], 500]];]
```

```
{1.843 Second, Null}
```

```
Timing[CircularVertices[1000];]
```

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

TO DO

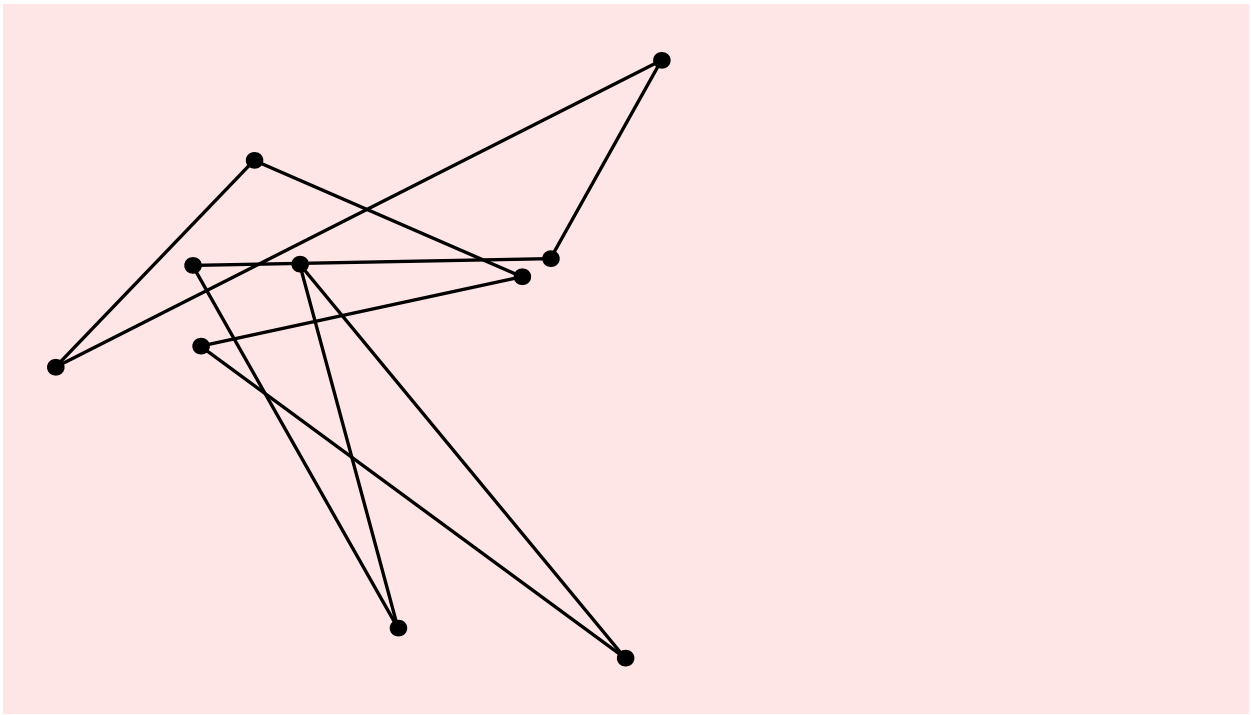
Directed version of the new ExactRandomGraph should also be provided.

RandomVertices

```
?RandomVertices
```

```
RandomVertices[g] assigns a random embedding to graph g.
```

```
ShowGraph[RandomVertices[Cycle[10]]]
```



```
- Graphics -
```

TIMING DISCUSSION

This function takes linear time and is extremely fast.

```
g = GridGraph[50, 50];
```

```
Timing[ RandomVertices[ g ]; ]
```

```
{0.031 Second, Null}
```

DegreeSequence

```
?DegreeSequence
```

```
DegreeSequence[g] gives the sorted degree sequence of graph g.
```

```
DegreeSequence[CompleteGraph[10]]
```

```
{9, 9, 9, 9, 9, 9, 9, 9, 9, 9}
```

```
DegreeSequence[Path[100]]
```

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

TIMING DISCUSSION

The function is linear in the size of graph and this is confirmed by the plot below.

The running times of the old and the new implementation are comparable.

```
g = GridGraph[40, 40]; h = DiscreteMath`OldCombinatorica`GridGraph[40, 40];
```

```
Timing[ DegreeSequence[ g ]; ]
```

```
{0.156 Second, Null}
```

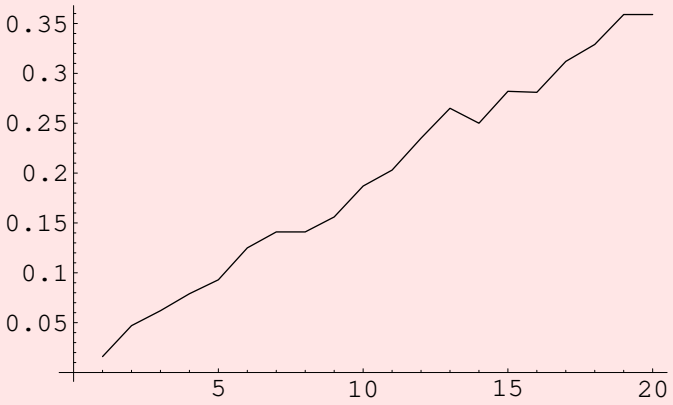
```
Timing[ DiscreteMath`OldCombinatorica`DegreeSequence[ h ]; ]
```

```
{0.672 Second, Null}
```

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

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

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

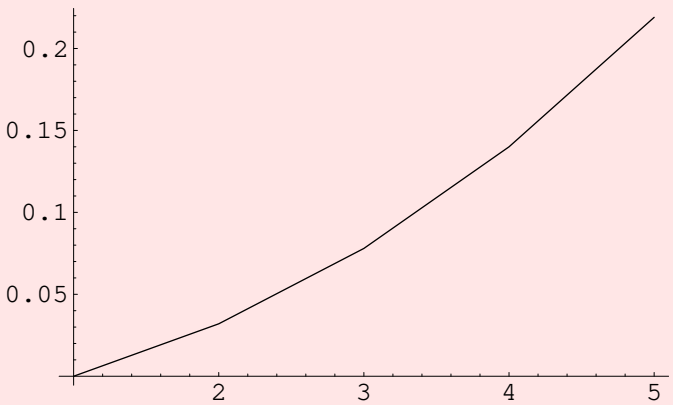



```
- Graphics -
```

```
gt = Table[ DiscreteMath`OldCombinatorica`GridGraph[20, 10 i], {i, 5}];
```

```
rt = Table[
   Timing[ DiscreteMath`OldCombinatorica`DegreeSequence[ gt[[i]]]; ], {i, 5}];
```

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




```
- Graphics -
```

GraphicQ

```
?GraphicQ
```

```
GraphicQ[s] yields True if the list of integers s is
  graphic, and thus represents a degree sequence of some graph.
```

```
GraphicQ[DegreeSequence[RandomGraph[100, .2]]]
```

```
True
```

```
GraphicQ[Range[100]]
```

```
False
```

TIMING DISCUSSION

GraphicQ and the new GraphicQ are almost identical and should take exactly the same amount of time.

I need to look at the code of this function more carefully, but it does seem that this should take linear time in the size of the sequence. From the plot this is not conclusive; in fact, the plot seems to indicate that the function might be taking quadratic time.

```
a = DiscreteMath`OldCombinatorica`RandomTree[400]; aa = RandomTree[400];
```

```
$RecursionLimit = 10000;
```

```
{Timing[GraphicQ[DegreeSequence[aa]];],
 Timing[DiscreteMath`OldCombinatorica`GraphicQ[
    DiscreteMath`OldCombinatorica`DegreeSequence[a]];]}
```

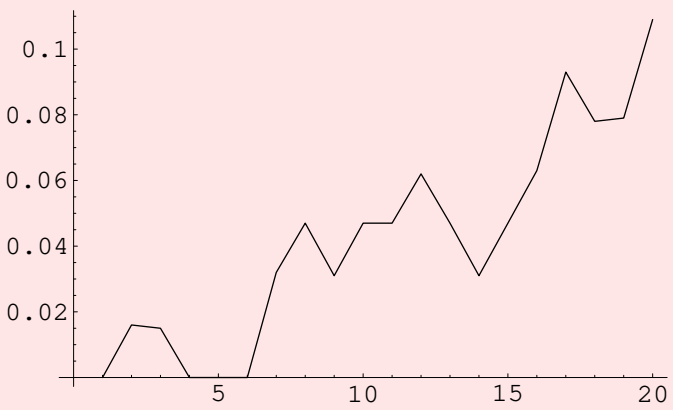
```
{{0.125 Second, Null}, {0.203 Second, Null}}
```

```
gt = Table [ DegreeSequence[ RandomTree[20 i]], {i, 20}];
```

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

```
{{0. Second, Null}, {0.016 Second, Null}, {0.015 Second, Null}, {0. Second, Null},
 {0. Second, Null}, {0. Second, Null}, {0.032 Second, Null}, {0.047 Second, Null},
 {0.031 Second, Null}, {0.047 Second, Null}, {0.047 Second, Null},
 {0.062 Second, Null}, {0.047 Second, Null}, {0.031 Second, Null},
 {0.047 Second, Null}, {0.063 Second, Null}, {0.093 Second, Null},
 {0.078 Second, Null}, {0.079 Second, Null}, {0.109 Second, Null}}
```

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




```
- Graphics -
```

RealizeDegreeSequence

```
?RealizeDegreeSequence
```

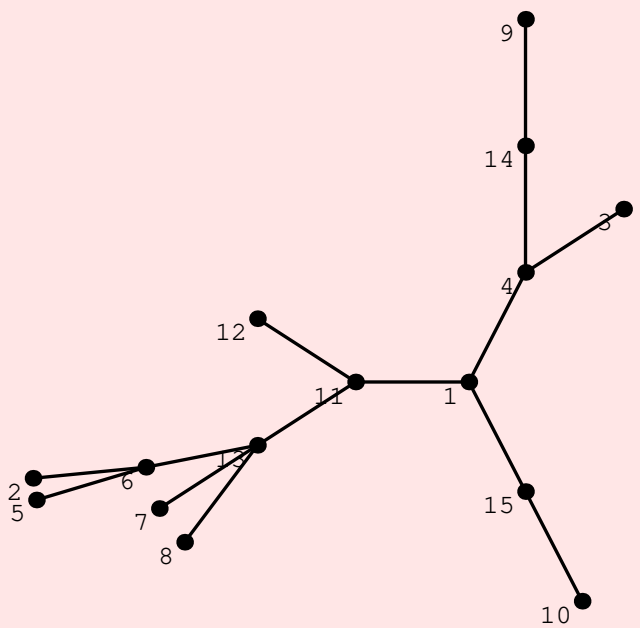
```
RealizeDegreeSequence[s] constructs
  a semirandom graph with degree sequence s.
```

NOTES

* I find this function a little strange. Why not just use GraphicQ's deterministic algorithm? Is the semi−randomgraph produced in this manner really useful? Does it produce a reasonably random Regular graph? Should we actually implement the algorithm in [Wor84] to generate a random regular graph?

* The new RealizeDegreeSequence is also a little ugly. Maybe its run−timecould be improved by using more sophisticated *Mathematica* constructs.

```
ShowGraph[s = RandomTree[15], VertexNumber -> On]
```

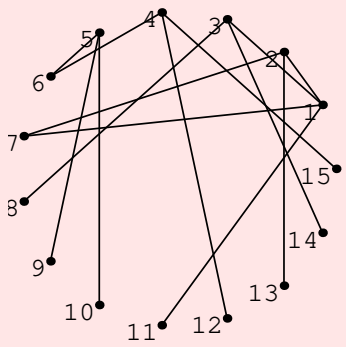



```
- Graphics -
```

```
seq = DegreeSequence[s]
```

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

```
ShowGraph[g = RealizeDegreeSequence[seq], VertexNumber -> On]
```




```
- Graphics -
```

```
DegreeSequence[g]
```

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

NOTES

* Note that the graph that we started with and the graph that resulted from taking its degree sequence and then running the new DegreeSequence are not isomorphic.

```
IsomorphicQ[s, g]
```

```
False
```

TIMING DISCUSSION

The function is quite slow and needs significant speedup. Also from the plot it is clear that the function is not linear in the size of the output it produces. The running times of the new and the old version are virtually identical. This function is definitely in need of significant speedup.

TO DO

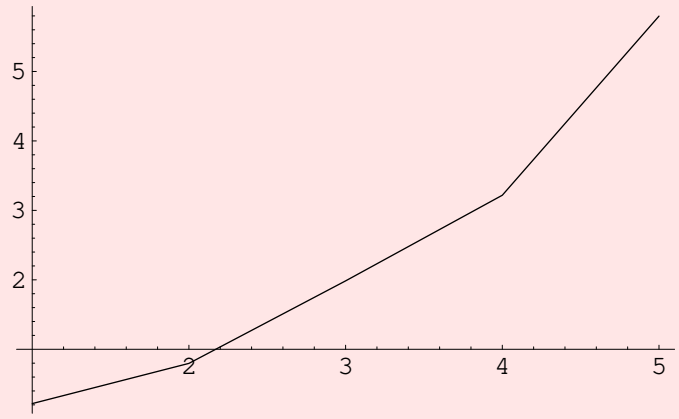
Speed up the RealizeDegreeSequence[...] function so that it is linear in the size of the output produced.

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

```
rt = Table[Timing[ RealizeDegreeSequence[ st[[i]] ]; ], {i, 5}]
```

```
{{0.218 Second, Null}, {0.797 Second, Null},
 {1.985 Second, Null}, {3.218 Second, Null}, {5.797 Second, Null}}
```

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




```
- Graphics -
```

```
st = Table[ DiscreteMath`OldCombinatorica`DegreeSequence[
   DiscreteMath`OldCombinatorica`GridGraph[10, 5 i]], {i, 5}] ; rt = Table[
  Timing[ DiscreteMath`OldCombinatorica`RealizeDegreeSequence[ st[[i]] ]; ],
  {i, 5}]
```

```
{{0.235 Second, Null}, {0.843 Second, Null},
 {1.907 Second, Null}, {3.531 Second, Null}, {5.328 Second, Null}}
```

RegularQ

```
?RegularQ
```

```
RegularQ[g] yields True if g is a regular graph.
```

```
RegularQ[CompleteGraph[10]]
```

```
True
```

```
RegularQ[CompleteGraph[10, 10]]
```

```
True
```

```
RegularQ[RandomTree[10]]
```

```
False
```

```
RegularQ[Cycle[10]]
```

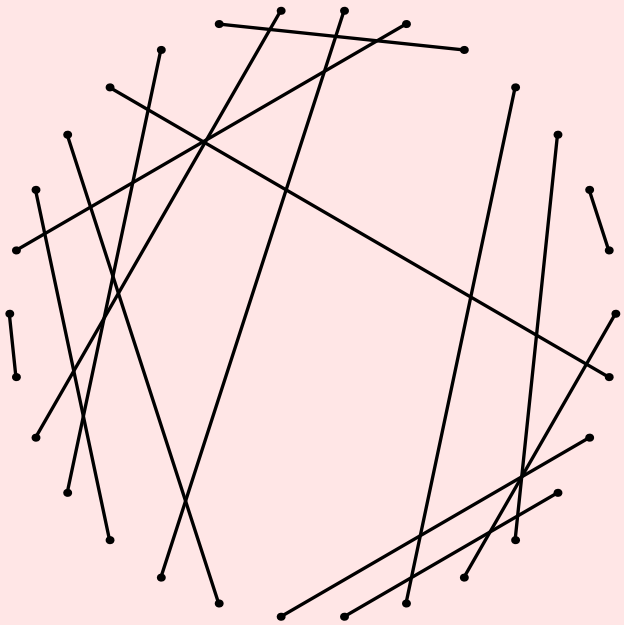
```
True
```

RegularGraph

```
?RegularGraph
```

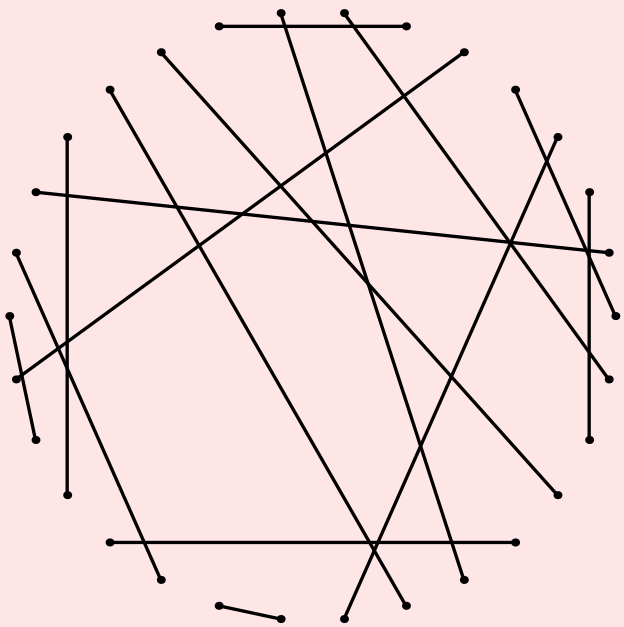
```
RegularGraph[k, n] constructs a semirandom
  k-regular graph on n vertices, if such a graph exists.
```

```
ShowGraph[RegularGraph[1, 30], VertexStyle -> Disc[Small]]
```



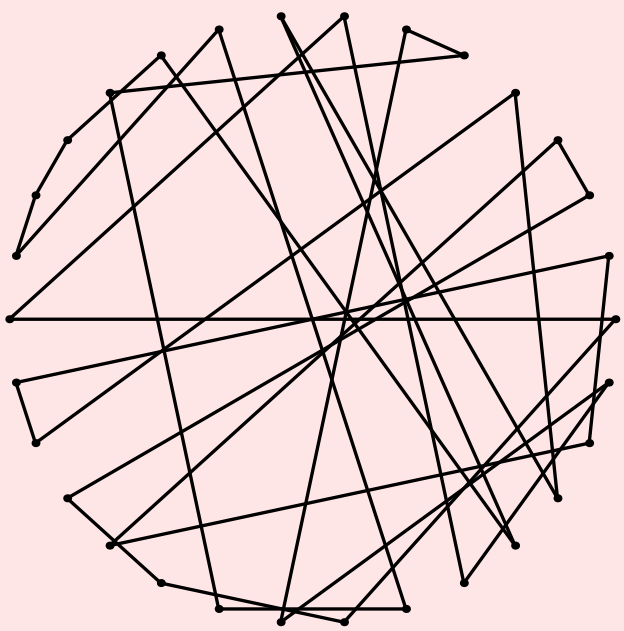
```
- Graphics -
```

```
ShowGraph[RealizeDegreeSequence[{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}], VertexStyle → Disc[Small]]
```



```
- Graphics -
```

```
ShowGraph[RegularGraph[2, 30], VertexStyle -> Disc[Small]]
```



```
- Graphics -
```

TIMING DISCUSSION

This function is essentially a call to RealizeDegreeSequence[...]. Hence, it has the same problems with running time as RealizeDegreeSequence does.

```
{Timing[DiscreteMath`OldCombinatorica`RegularGraph[10, 100];],
 Timing[RegularGraph[10, 100];]}
```

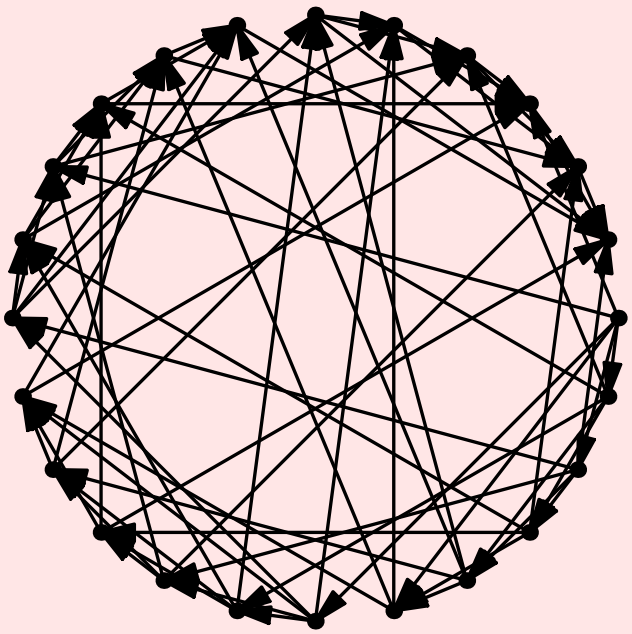
```
{{0.86 Second, Null}, {0.781 Second, Null}}
```

MakeGraph

```
?MakeGraph
```

```
MakeGraph[v, f] constructs the graph whose vertices correspond to
  v and edges between pairs of vertices x and y in v for which
  the binary relation defined by the boolean function f is true.
```

```
ShowGraph[g = MakeGraph[Permutations[{1, 2, 3, 4}],
    (Count[#1 - #2, 0] == 2 && (Inversions[#1] > Inversions[#2])) &]];
```

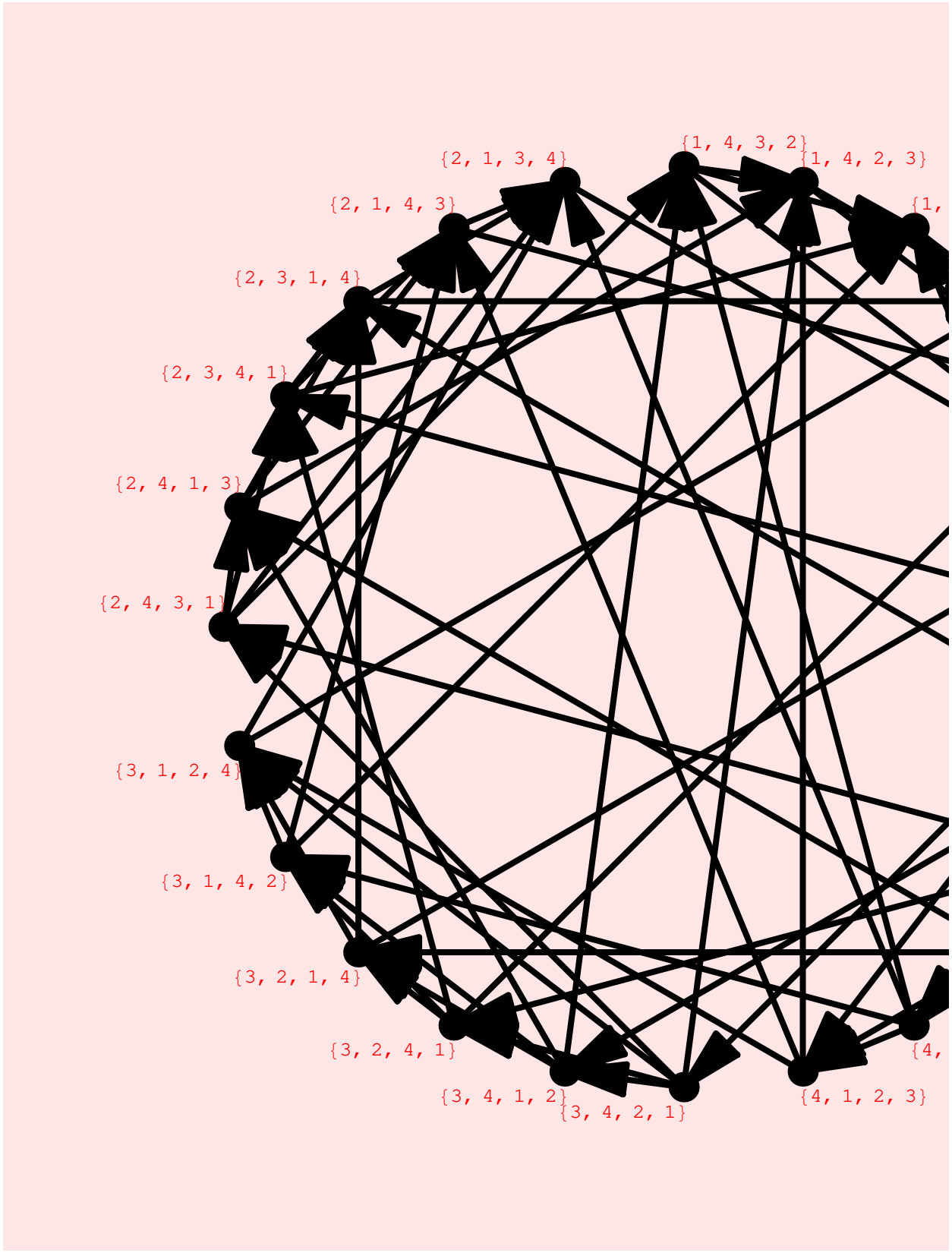


```
p = Permutations[ {1, 2, 3, 4}];
```

```
p1 = Map[ Text[#, {0.05, 0.02}] &, Cases[p, {1, ___}]];
p2 = Map[ Text[#, {-0.05, 0.02}] &, Cases[p, {2, ___}]];
p3 = Map[ Text[#, {-0.05, -0.02}] &, Cases[p, {3, ___}]];
p4 = Map[ Text[#, {0.05, -0.02}] &, Cases[p, {4, ___}]];
```

```
g = SetVertexLabels[g, Join[p1, p2, p3, p4] ];
```

```
ShowGraph[ g , ImageSize -> 600, VertexLabelColor -> Red, PlotRange -> Large[0.2]
```


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

```
- Graphics -
```

IntervalGraph

```
? IntervalGraph
```

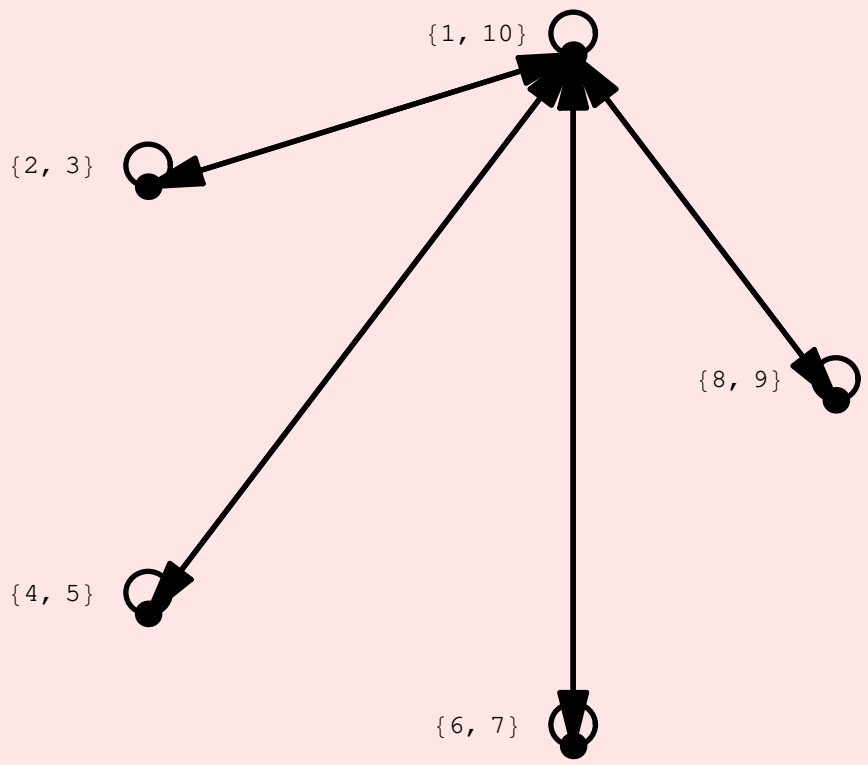
```
IntervalGraph[l] constructs the
  interval graph defined by the list of intervals l.
```

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

```
s = IntervalGraph[l];
```

```
s = SetVertexLabels[s, Map[ Text[#, {-0.09, 0.02}] &, l]];
```

```
ShowGraph[ s , PlotRange -> Large[.4], ImageSize -> 400]
```



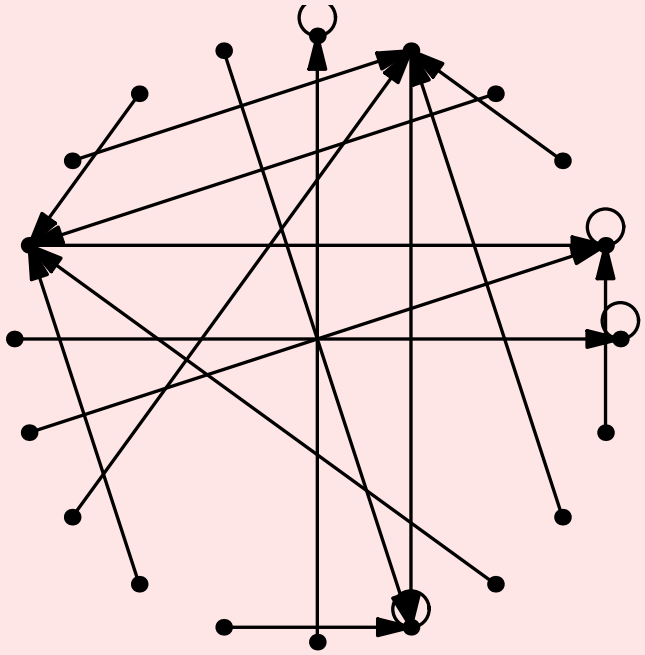

```
- Graphics -
```

FunctionalGraph

```
?FunctionalGraph
```

```
FunctionalGraph[f, v] takes a set v and a function f from v to v and
  constructs a directed graph with vertex set v and edges (x, f(x))
  for each x in v. FunctionalGraph[f, s, v] constructs a graph with
  vertex set v and edge set (x, f(e, x)) for every x in v and e in
  s. An option called Type that takes on the values Directed and
  Undirected is allowed. Type -> Directed is default, while Type ->
  Undirected returns the corresponding underlying undirected graph.
```

```
ShowGraph[FunctionalGraph[Mod[(#^2 + 2 * #), 20] &, Table[i, {i, 0, 19}]]]
```



```
- Graphics -
```