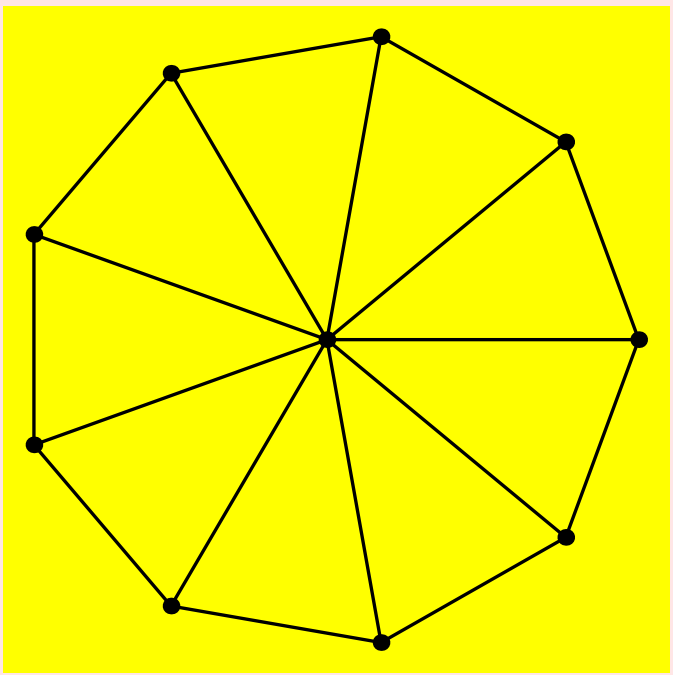
3. Representing Graphs

```
t = Wheel[10]; ShowGraph[t, Background -> Yellow]
```



```
- Graphics -
```

Edges

```
?Edges
```

```
Edges[g] gives the list of edges in g. Edges[g,
  All] gives the edges of g along with the graphics options
  associated with each edge. Edges[g, EdgeWeight] returns
  the list of edges in g along with their edge weights.
```

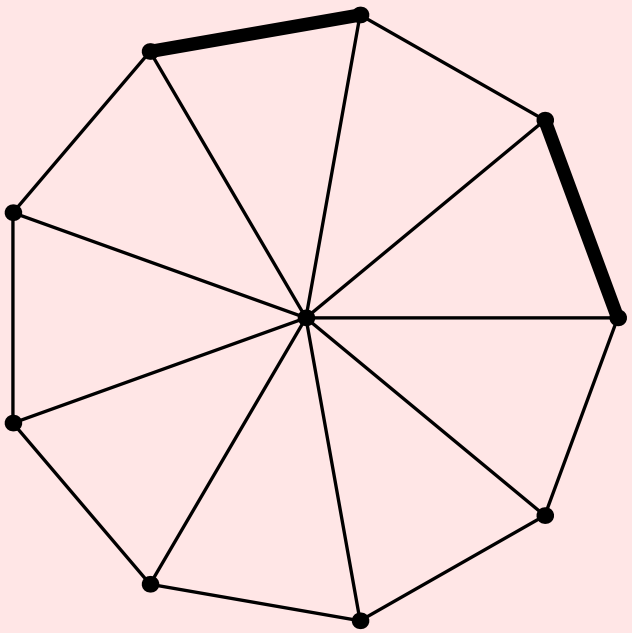
```
Edges[t]
```

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

```
Edges[t1 = SetGraphOptions[t, {{{1, 9}, {2, 3}, EdgeStyle -> Fat}}], All]
```

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

```
ShowGraph[t1]
```



```
- Graphics -
```

```
Edges[t2 = SetGraphOptions[t, {{{1, 9}, {2, 3}, EdgeStyle -> Fat}}]]
```

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

```
Edges[t, EdgeWeight]
```

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

```
?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.
```

```
k = SetEdgeWeights[t,
  WeightingFunction -> RandomInteger, WeightRange -> {10, 20}]
```

```
-Graph:<18, 10, Undirected>-
```

```
Edges[k, EdgeWeight]
```

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

Vertices

```
?Vertices
```

```
Vertices[g] gives the embedding of graph g, that is, the coordinates of
  each vertex in the plane. Vertices[g, All] gives the embedding of
  the graph along with graphics options associated with each vertex.
```

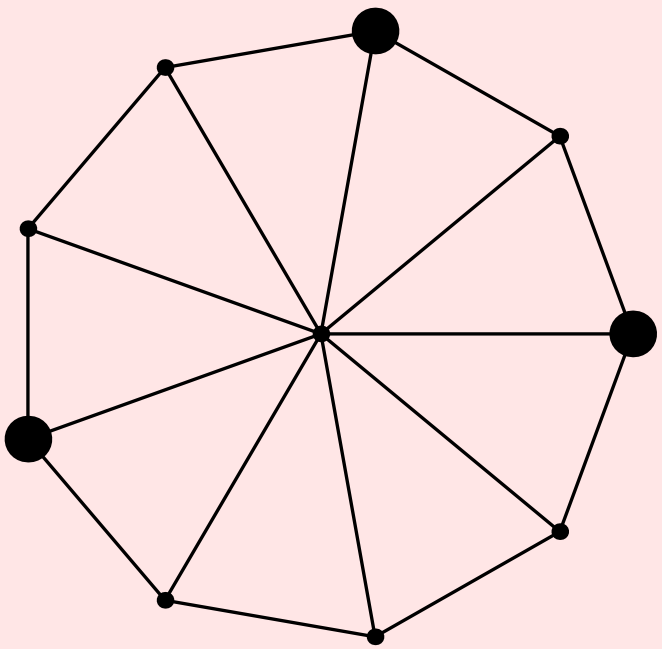
```
Vertices[t]
```

```
{{0.766044, 0.642788}, {0.173648, 0.984808}, {-0.5, 0.866025},
 {-0.939693, 0.34202}, {-0.939693, -0.34202}, {-0.5, -0.866025},
 {0.173648, -0.984808}, {0.766044, -0.642788}, {1., 0}, {0, 0}}
```

```
Vertices[
 t3 = SetGraphOptions[t, {{2, 5, 9, VertexStyle -> Disc[Large]}}], All]
```

```
{{{0.766044, 0.642788}}, {{0.173648, 0.984808}, VertexStyle → Disc[Large]},
 {{-0.5, 0.866025}}, {{-0.939693, 0.34202}},
 {{-0.939693, -0.34202}, VertexStyle → Disc[Large]},
 {{-0.5, -0.866025}}, {{0.173648, -0.984808}}, {{0.766044, -0.642788}},
 {{1., 0}, VertexStyle → Disc[Large]}, {{0, 0}}}
```

```
ShowGraph[t3]
```

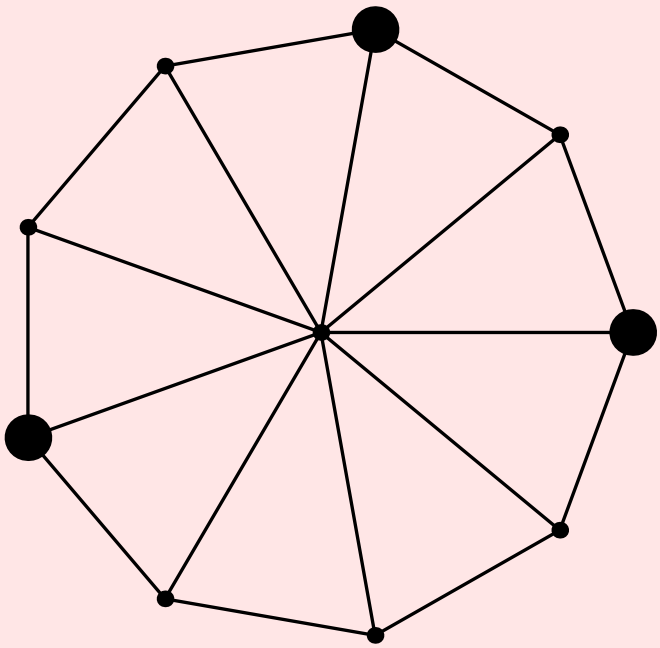


```
- Graphics -
```

```
Vertices[t4 = SetGraphOptions[t, {{2, 5, 9, VertexStyle -> Disc[Large]}}]]
```

```
{{0.766044, 0.642788}, {0.173648, 0.984808}, {-0.5, 0.866025},
 {-0.939693, 0.34202}, {-0.939693, -0.34202}, {-0.5, -0.866025},
 {0.173648, -0.984808}, {0.766044, -0.642788}, {1., 0}, {0, 0}}
```

```
ShowGraph[t4]
```



```
- Graphics -
```

V

```
?V
```

```
V[g] gives the order or number of vertices of the graph g.
```

```
V[t]
```

```
10
```

```
V[DeleteVertices[t, {1}]]
```

```
9
```

```
V[RandomGraph[20, .4]]
```

```
20
```

M

```
?M
```

```
M[g] gives the number of edges in the graph g. M[g, Directed] is
  obsolete because M[g] works for directed as well as undirected graphs.
```

```
M[t]
```

```
18
```

```
M[AddEdges[t, {{{1, 2}}}]]
```

```
19
```

NOTES

There are no options for M (unlike for the old version of M, that has a Directed/Undirected option). Multiple edges get counted separately and self−loopsget counted once for each loop.

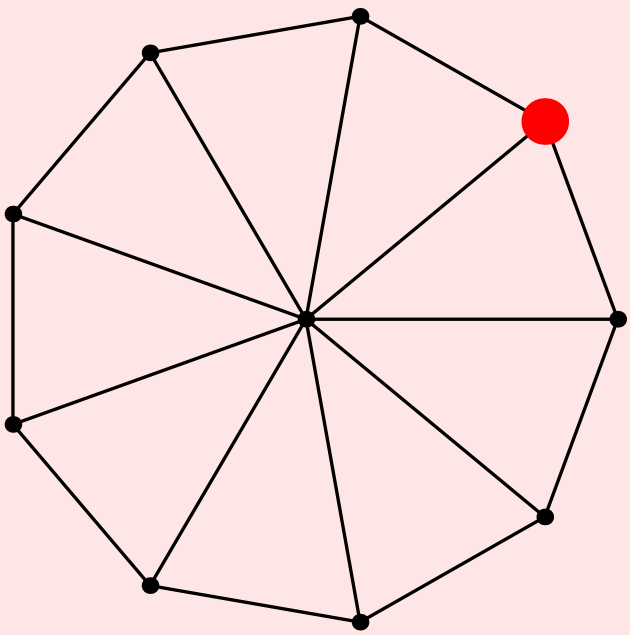
ChangeVertices

```
?ChangeVertices
```

```
ChangeVertices[g, v] replaces the vertices of graph g with the
  vertices in the given list v. v can have the form {{x1, y1}, {x2,
  y2}, ...} or the form {{{x1, y1}, gr1}, {{x2, y2}, gr2}, ...},
  where {x1, y1}, {x2, y2}, ... are coordinates of points and
  gr1, gr2, ... are graphics information associated with vertices.
```

```
s = SetGraphOptions[t, {{1, VertexColor -> Red, VertexStyle -> Disc[Large]}}];
```

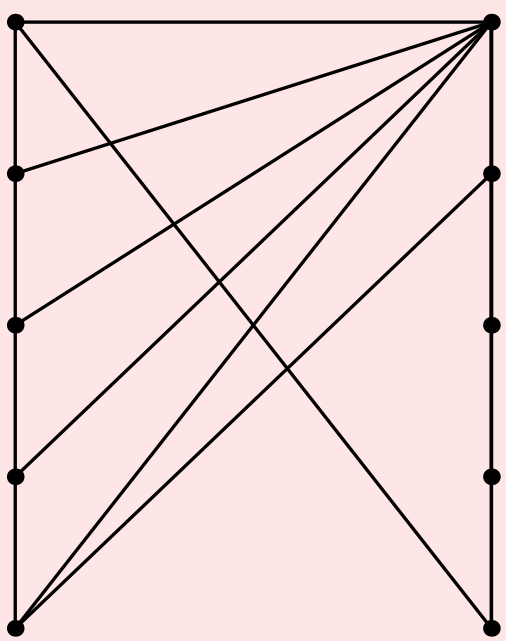
```
ShowGraph[s]
```



```
- Graphics -
```

```
s = ChangeVertices[t, Vertices[CompleteGraph[5, 5], All]];
```

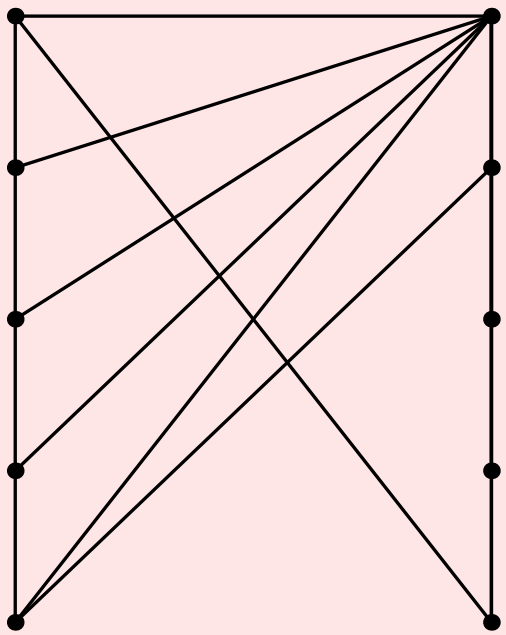
```
ShowGraph[s]
```



```
- Graphics -
```

```
s = ChangeVertices[t, Vertices[CompleteGraph[5, 5]]];
```

```
ShowGraph[s]
```



```
- Graphics -
```

NOTES

The above example shows us a new 2−leveledembedding of a wheel. Difficult to recognize that the graph is a wheel. ChangeVertices is therefore a good function to use to produce new embeddings of graphs.

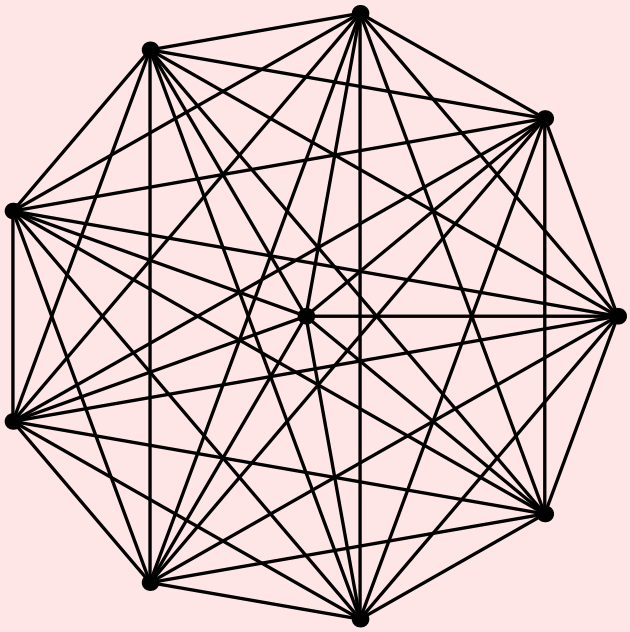
ChangeEdges

```
?ChangeEdges
```

```
ChangeEdges[g, e] replaces the edges of graph g with the
  edges in e. e can have the form {{s1, t1}, {s2, t2}, ...}
  or the form { {{s1, t1}, gr1}, {{s2, t2}, gr2}, ...}, where
  {s1, t1}, {s2, t2}, ... are endpoints of edges and gr1,
  gr2, ... are graphics information associated with edges.
```

```
s = ChangeEdges[t, Edges[CompleteGraph[10], All]];
```

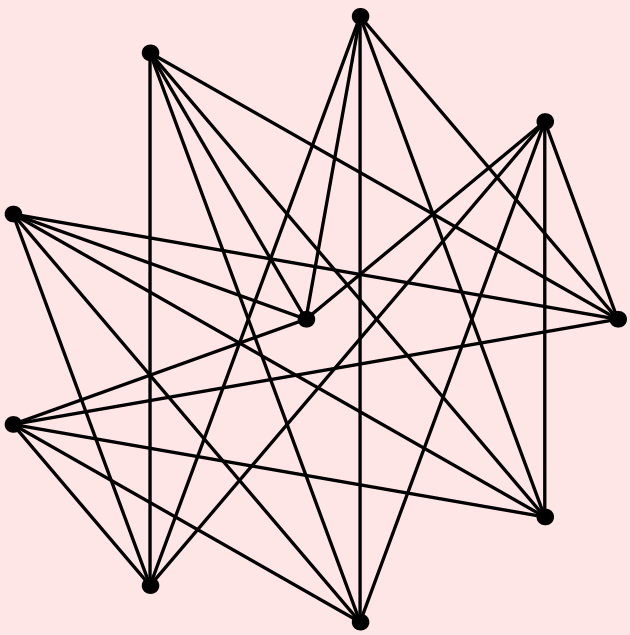
```
ShowGraph[s]
```



```
- Graphics -
```

```
s = ChangeEdges[t, Edges[CompleteGraph[5, 5], All]];
```

```
ShowGraph[s]
```



```
- Graphics -
```

NOTES

ChangeEdges is another way of producing new embeddings of graphs.

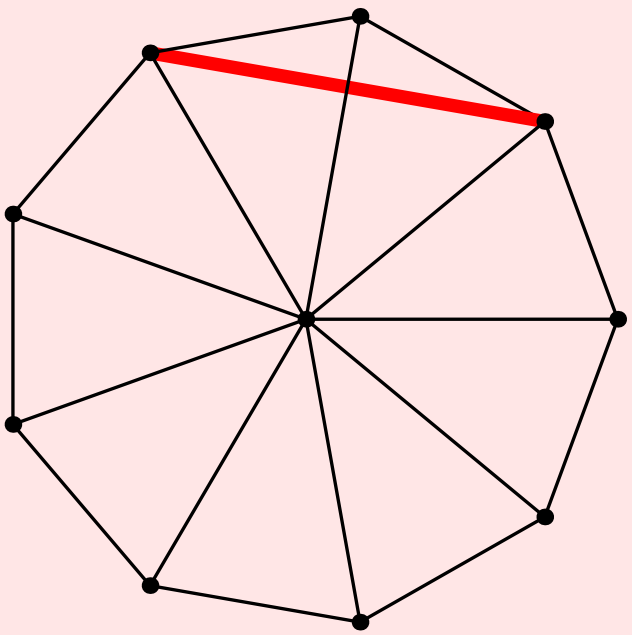
AddEdges

```
?AddEdges
```

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

```
s = AddEdges[t, {{{1, 3}, EdgeStyle -> Fat, EdgeColor -> Red}}];
```

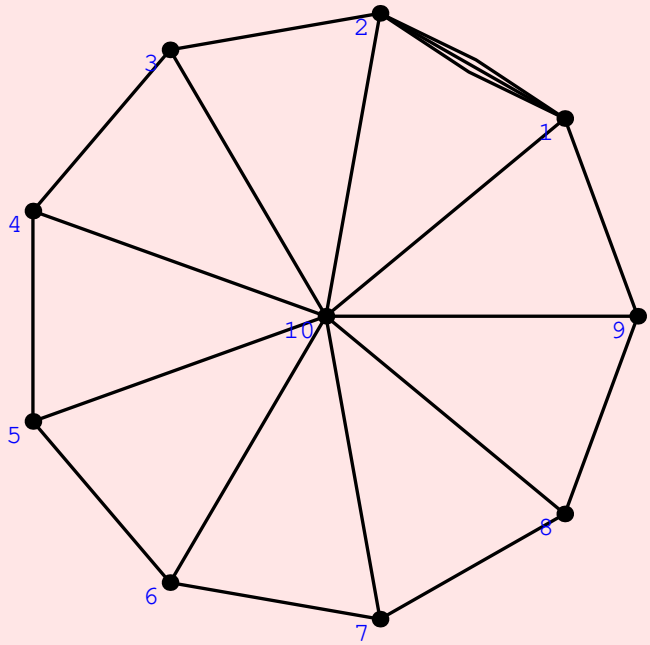
```
ShowGraph[s]
```



```
- Graphics -
```

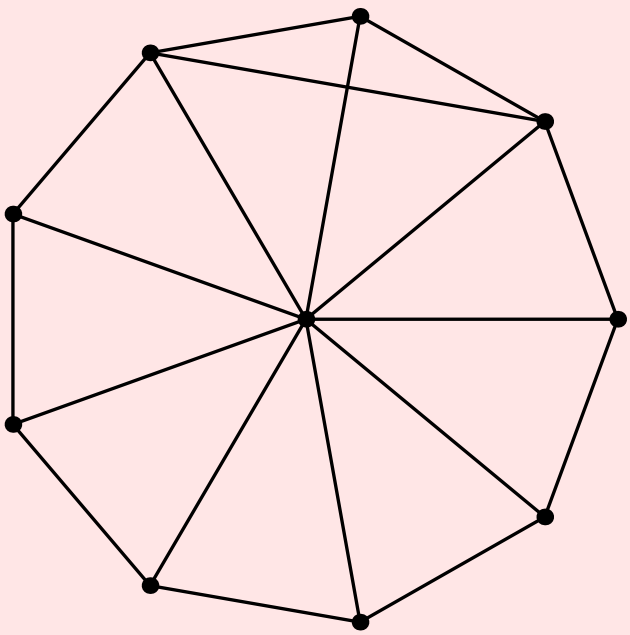
```
s = AddEdges[t, {{1, 2}, {2, 1}}];
```

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




```
- Graphics -
```

```
ShowGraph[AddEdges[t, {1, 3}]]
```



```
- Graphics -
```

NOTES

* An edge can be addded along with any associated graphics information.

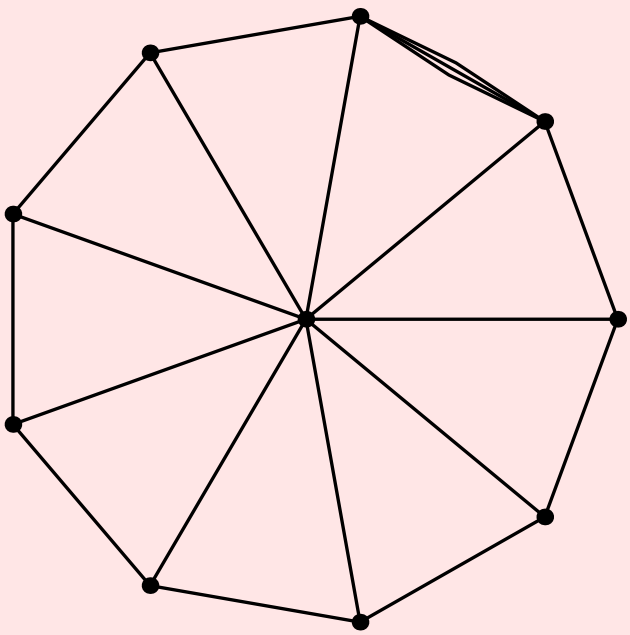
* Adding an edge that already exists results in multiple edges.

DeleteEdges

```
?DeleteEdges
```

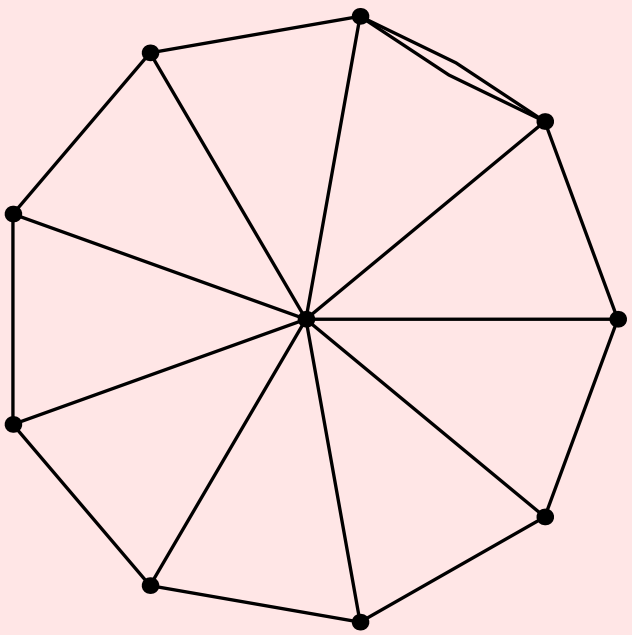
```
DeleteEdges[g, edgeList] gives graph g minus list of edges edgeList. If
  g is undirected then the edges in edgeList are treated as undirected
  edges, or otherwise they are treated as directed edges. If there
  are multiple edges that qualify, then only one edge is deleted.
  DeleteEdges[g, edgeList, All] will delete all edges that qualify.
  If only one edge is to be deleted, then edgeList can have the form
  {s, t}, or otherwise it has the form {{s1, t1}, {s2, t2}, ...}.
```

```
ShowGraph[s]
```



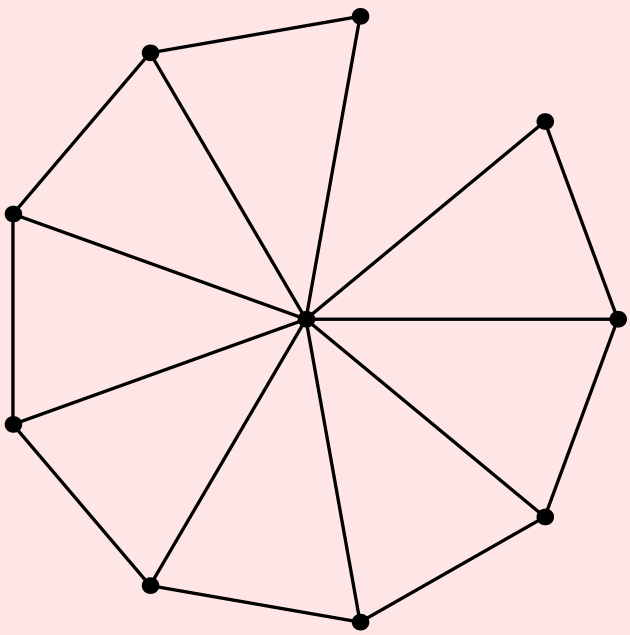
```
- Graphics -
```

```
ShowGraph[DeleteEdges[s, {{1, 2}}]]
```



```
- Graphics -
```

```
ShowGraph[DeleteEdges[s, {{1, 2}}, All]]
```



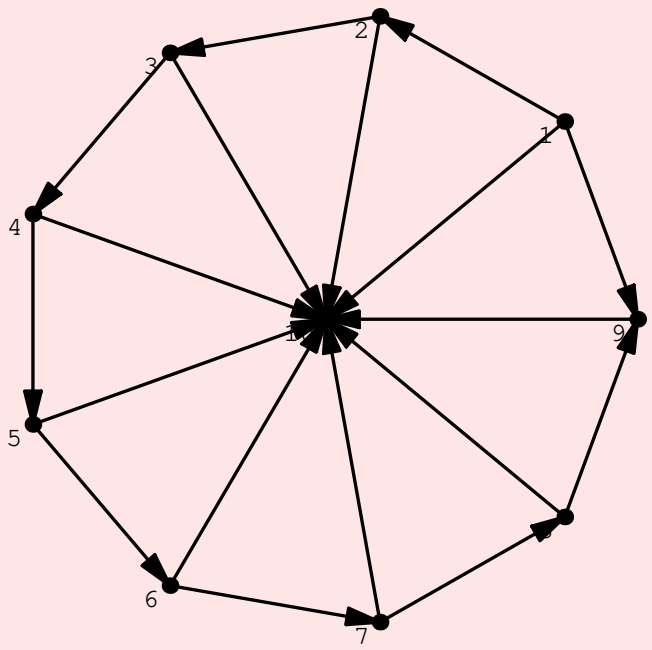
```
- Graphics -
```

NOTES

* There are three edges between 1 and 2 in the graph s and deleting {1,2} deletes only one edge and leaves two other edges connecting 1 and 2 behind. The outcome is different when ALL is provided as the third argument.

```
s = SetGraphOptions[t, EdgeDirection -> On];
```

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

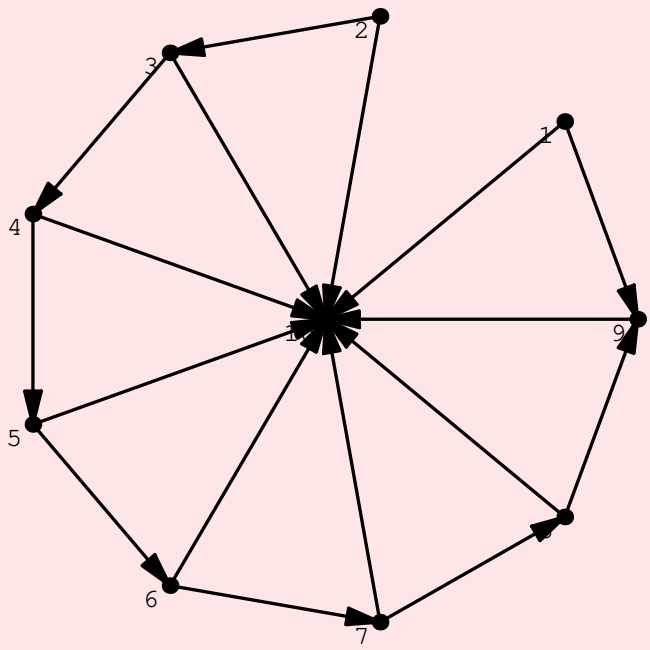



```
- Graphics -
```

NOTES

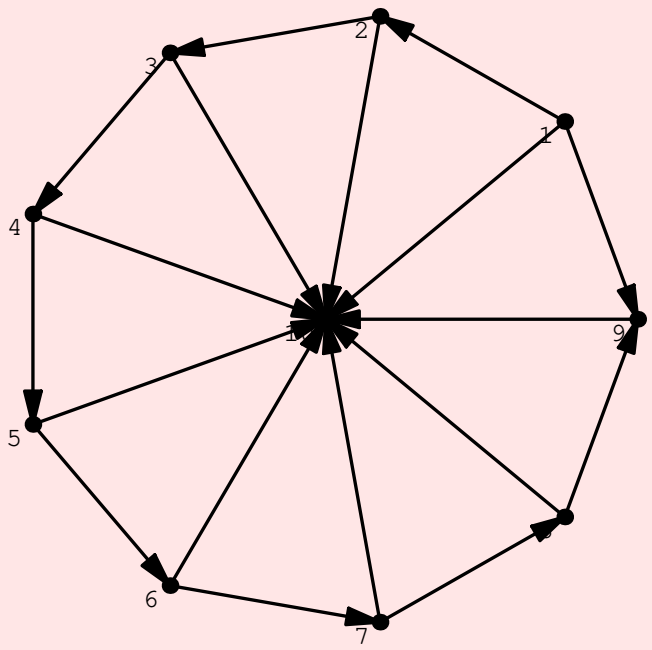
* If the graph is directed then the edges in the given edge list are also treated as directed edges. Below, the directed edge {1, 2} is deleted from s. Further below, nothing is deleted since {2,1} is not a directed edge in s.

```
ShowGraph[DeleteEdges[s, {{1, 2}}], VertexNumber -> On]
```



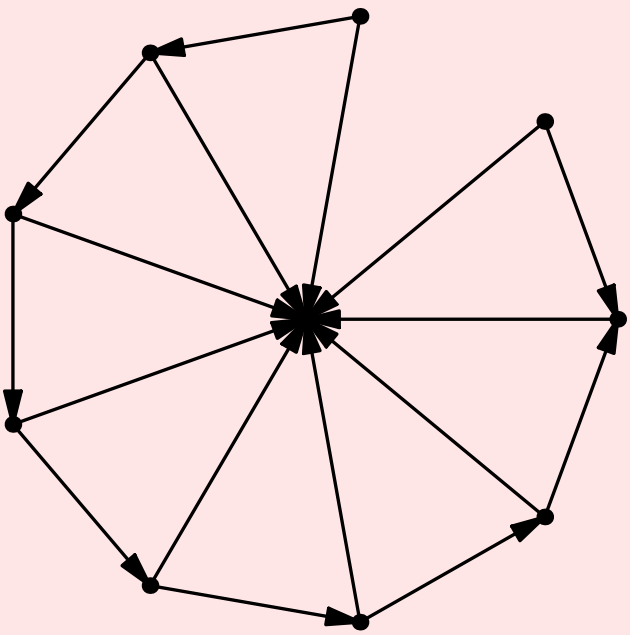

```
- Graphics -
```

```
ShowGraph[DeleteEdges[s, {{2, 1}}], VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[DeleteEdges[s, {1, 2}]]
```



```
- Graphics -
```

TO DO

* DeleteEdges should also permit the user to use the function by simply typing DeleteEdges[g, {1,2}] if the user is deleting only one edge.

TIMING DISCUSSION

With and without the ALL argument, deleting d edges from an m–edgegraph takes θ(md) time. In the following expriment, m varies from 110 to 350 in steps of 10 and d is m/2. The expectation is that the running time will show a quadratic increase in m. The plot below does not show this conclusively. The Select–MemberQcombination seems to work quite well and keep the constant factors very small. This is shown by using Fit below to obtain a quadratic that fits the data. The coefficient of x^2 is rather small.

```
?ExactRandomGraph
```

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

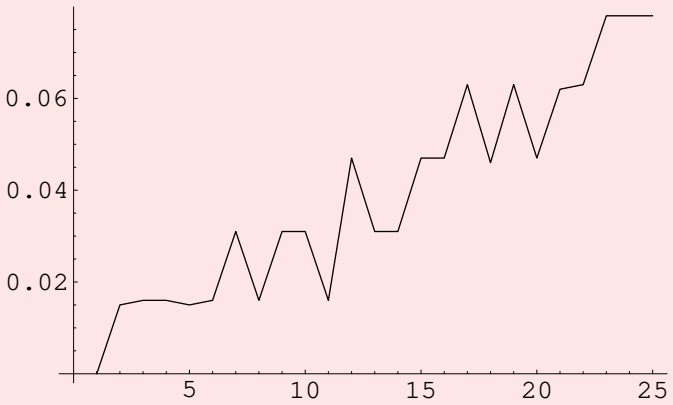
```
gt = Table[ExactRandomGraph[50, 100 + 10 i], {i, 25}];
```

```
et = Table[
   Edges[gt[[i]]] [[RandomKSubset[Range[100 + 10 i], 50 + 5 i]]], {i, 25}];
```

```
rt = Table[Timing[DeleteEdges[gt[[i]], et[[i]]];][[1]], {i, 25}]
```

```
{0. Second, 0.015 Second, 0.016 Second, 0.016 Second, 0.015 Second,
 0.016 Second, 0.031 Second, 0.016 Second, 0.031 Second, 0.031 Second,
 0.016 Second, 0.047 Second, 0.031 Second, 0.031 Second, 0.047 Second,
 0.047 Second, 0.063 Second, 0.046 Second, 0.063 Second, 0.047 Second,
 0.062 Second, 0.063 Second, 0.078 Second, 0.078 Second, 0.078 Second}
```

```
ListPlot[Map[First, rt], PlotJoined -> True]
```




```
- Graphics -
```

```
Fit[ Map[First, rt], {1, x, x^2}, x]
```

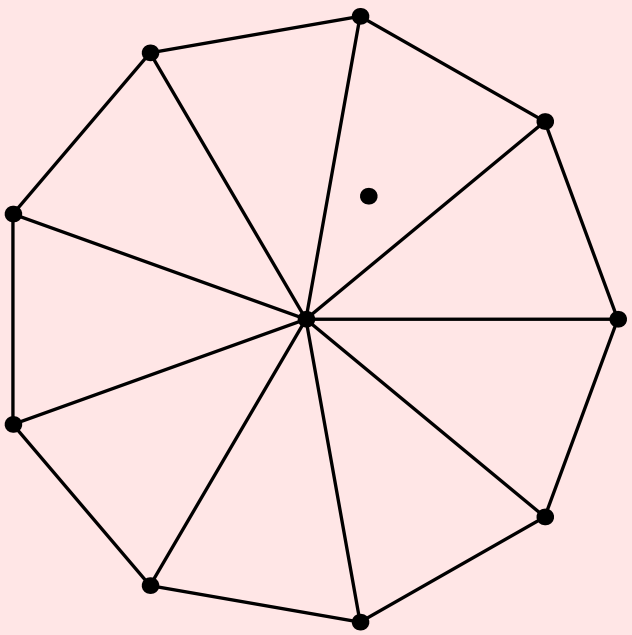
$$0.00672435 + 0.00168886\,x + 0.0000483278\,x^2$$

AddVertices

```
?AddVertices
```

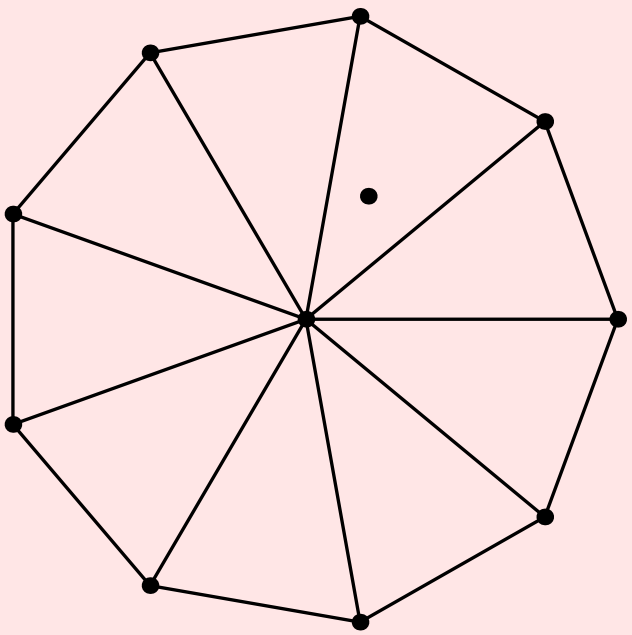
```
AddVertices[g, n] adds n disconnected vertices to graph g. AddVertices[
  g, vList] adds vertices in  vList to g. vList contains embedding
  and graphics information and can have the form {x, y} or {{x1, y1},
  {x2, y2}...} or the form {{{x1, y1}, g1}, {{x2, y2}, g2},...},
  where {x, y}, {x1, y1},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

```
ShowGraph[AddVertices[t, {.2, .4}]]
```



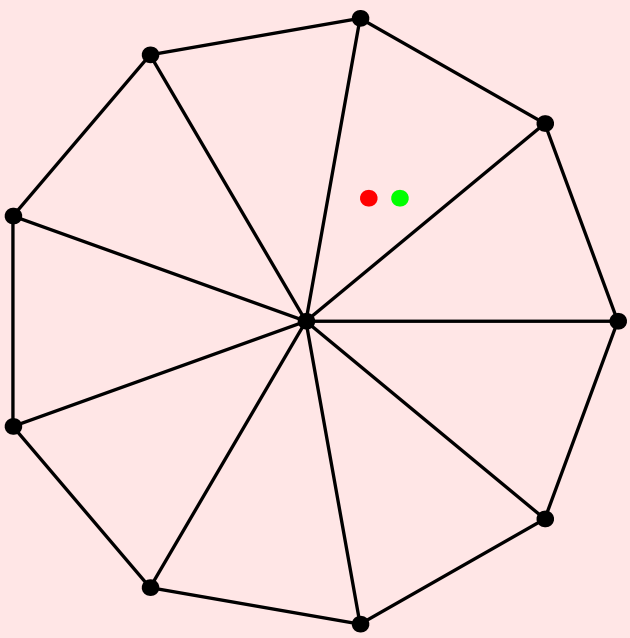
```
- Graphics -
```

```
ShowGraph[AddVertices[t, {{{.2, .4}}}]]
```



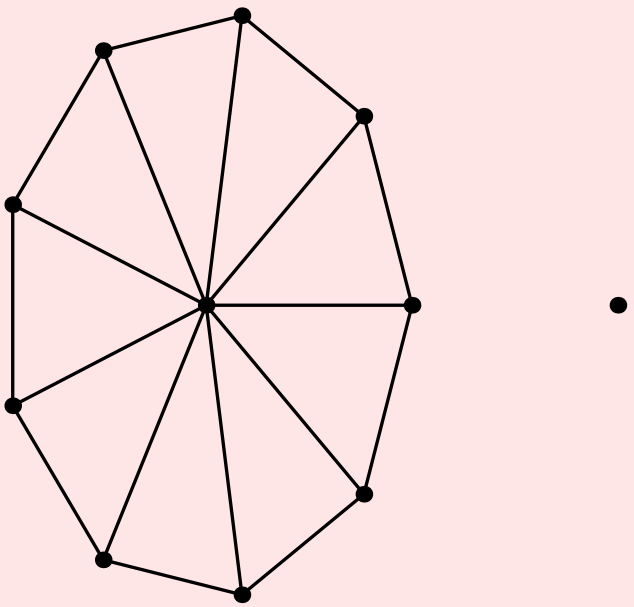
```
- Graphics -
```

```
ShowGraph[AddVertices[t,
  {{{.2, .4}, VertexColor → Red}, {{.3, .4}, VertexColor → Green}}]]
```



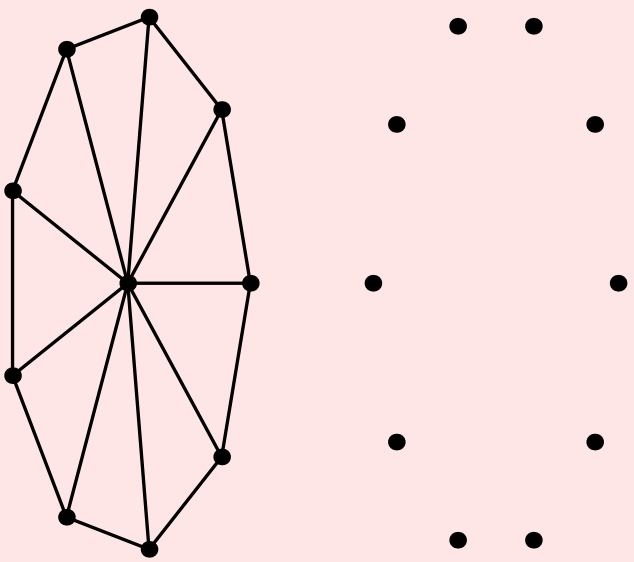
```
- Graphics -
```

```
ShowGraph[AddVertices[t, 1]]
```



```
- Graphics -
```

```
ShowGraph[AddVertices[t, 10]]
```

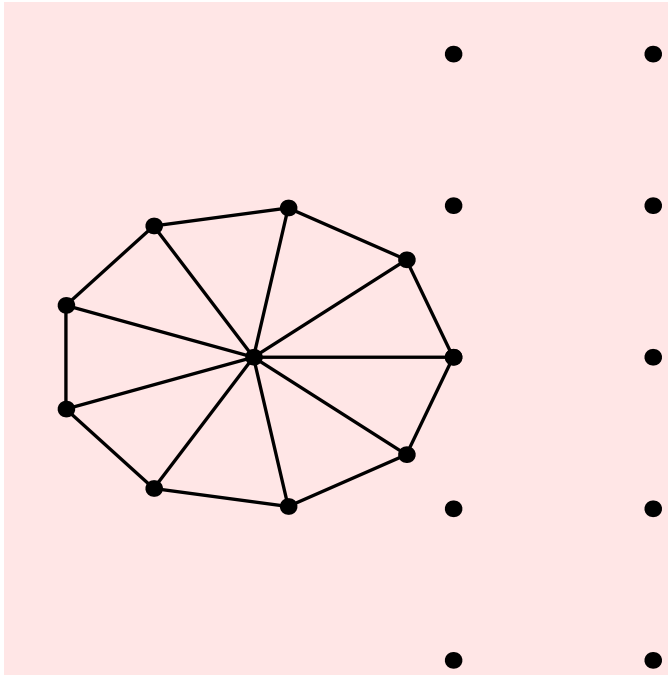


```
- Graphics -
```

NOTES

* Below, 10 vertices, whose embedding is derived from the embedding of K_{5, 5} are added to t. In this particular example, one of the newly added vertices coincides with a vertex in t and hence seems to be missing.

```
ShowGraph[AddVertices[t, Vertices[CompleteGraph[5, 5], All]]]
```



```
- Graphics -
```

DeleteVertices

```
?DeleteVertices
```

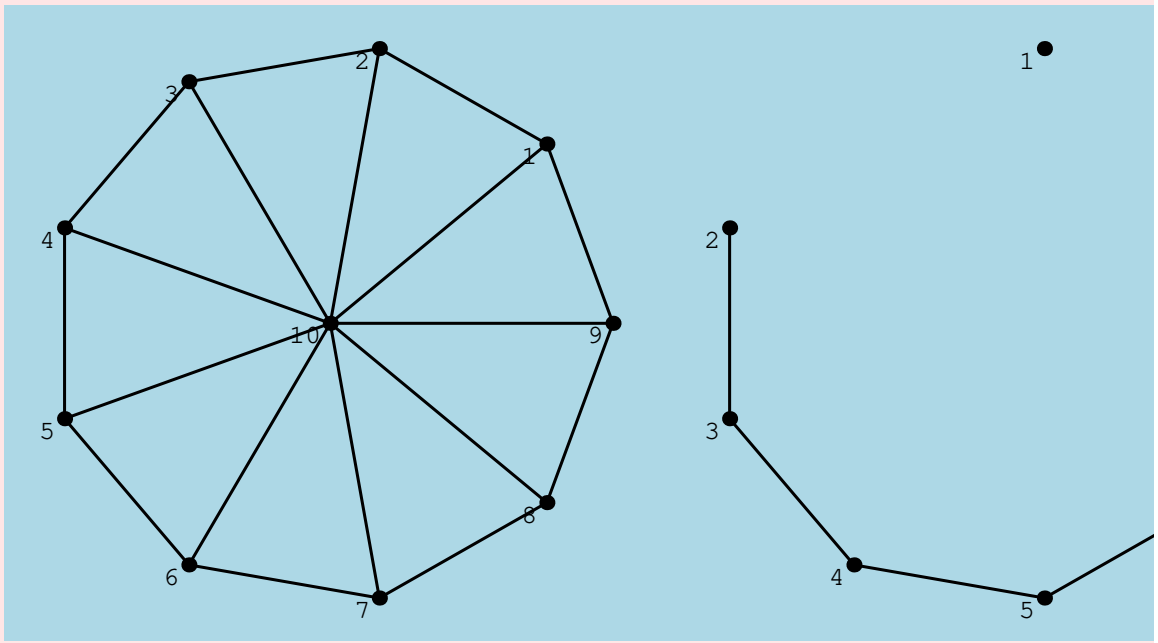
```
DeleteVertices[g, vList] deletes vertices in vList from graph g. vList
  has the form {i, j, ...}, where i, j, ... are vertex numbers.
```

```
s = DeleteVertices[t, {1, 3, 10}];
```

```
p1 = ShowGraph[t, VertexNumber -> On, Graphics];
```

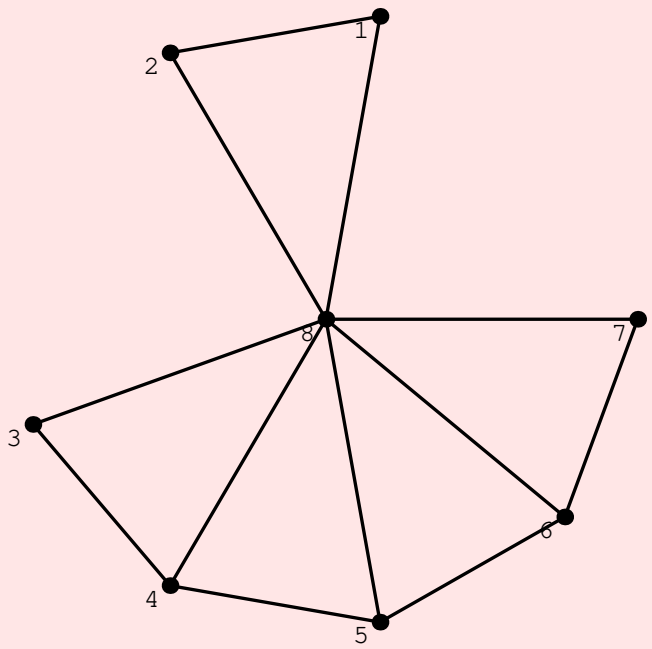
```
p2 = ShowGraph[s, VertexNumber -> On, Graphics];
```

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




```
- GraphicsArray -
```

```
ShowGraph[DeleteVertices[DeleteVertices[DeleteVertices[t, {1}], {3}], {10}],
 VertexNumber -> On]
```




```
- Graphics -
```

NOTES

* Note how deleting vertices 1, 3, and 10 successively gives a different graph as compared to the one obtained by deleting the set of vertices {1, 3, 10}. This because vertices are renumbered after each deletion.

Spectrum

```
? Spectrum
```

```
Spectrum[g] gives the eigenvalues of graph g.
```

Spectrum[t]

$$\left\{-1,\ -1,\ 1-\sqrt{10},\ 1+\sqrt{10},\ \frac{1}{\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3}}+\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3},\right.$$
$$\frac{1}{\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3}}+\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3},$$
$$-\frac{1-i\sqrt{3}}{2^{2/3}\left(-1+i\sqrt{3}\right)^{1/3}}-\frac{1}{2}\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3}\left(1+i\sqrt{3}\right),$$
$$-\frac{1-i\sqrt{3}}{2^{2/3}\left(-1+i\sqrt{3}\right)^{1/3}}-\frac{1}{2}\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3}\left(1+i\sqrt{3}\right),$$
$$-\frac{1}{2}\left(1-i\sqrt{3}\right)\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3}-\frac{1+i\sqrt{3}}{2^{2/3}\left(-1+i\sqrt{3}\right)^{1/3}},$$
$$\left.-\frac{1}{2}\left(1-i\sqrt{3}\right)\left(\frac{1}{2}\left(-1+i\sqrt{3}\right)\right)^{1/3}-\frac{1+i\sqrt{3}}{2^{2/3}\left(-1+i\sqrt{3}\right)^{1/3}}\right\}$$

Spectrum[GraphUnion[Cycle[4], CompleteGraph[1]]]

$$\{-2,\ 0,\ 0,\ 0,\ 2\}$$

Spectrum[Star[5]]

$$\{-2,\ 0,\ 0,\ 0,\ 2\}$$

Spectrum[CompleteGraph[3, 4]]

$$\left\{0,\ 0,\ 0,\ 0,\ 0,\ -2\sqrt{3},\ 2\sqrt{3}\right\}$$

ToAdjacencyLists

```
? ToAdjacencyLists
```

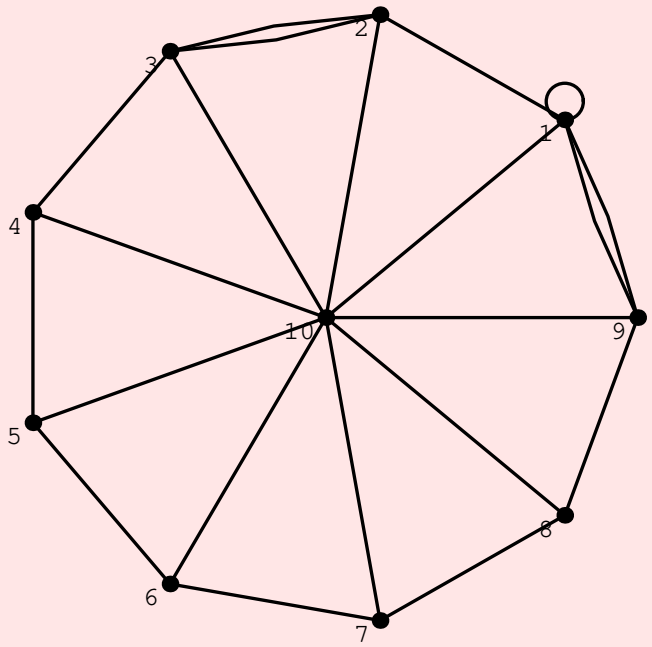
```
ToAdjacencyLists[g] constructs an adjacency list representation for
  graph g. It allows an option called Type that takes on values All or
  Simple. Type -> All is the default setting of the option, and this
  permits self-loops and multiple edges to be reported in the adjacency
  lists. Type -> Simple deletes self-loops and multiple edges from
  the constructed adjacency lists. ToAdjacencyLists[g, EdgeWeight]
  returns an adjacency list reprsentation along with edge weights.
```

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

```
t = AddEdges[t, {{{1, 9}}}];
```

```
t = AddEdges[t, {{{2, 3}}}];
```

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




```
- Graphics -
```

```
l0 = ToAdjacencyLists[t]
```

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

```
ToAdjacencyLists[t, EdgeWeight]
```

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

NOTES

* t is a graph that has a self−loop{1, 1} and multiple edges: two edges between 2 and 3 and two between 1 and 9.Note that in the above example the self−loop{1, 1} shows up as the 1 in 1's adjacency list. The multiple edges {2, 3} show up with two 3's in 2's adjacency list and two 2's in 3's adjacency list and the multiple edges {1, 9} show up similarly.

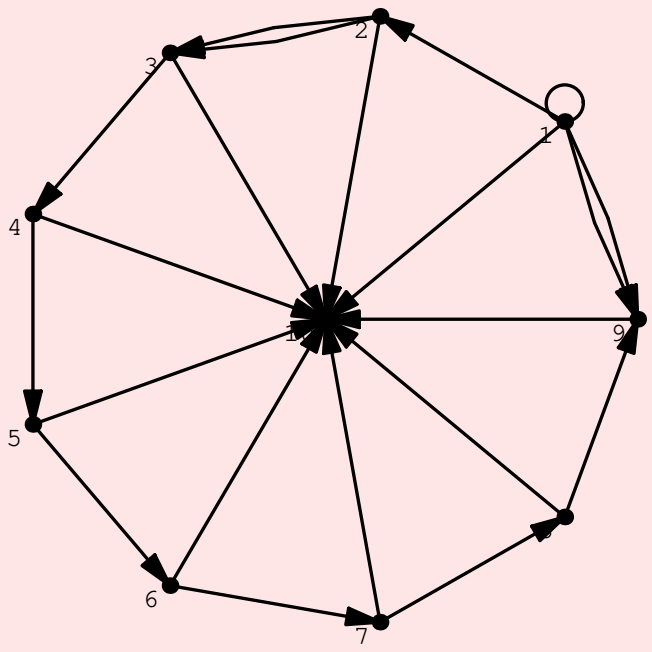
* ToAdjacencyLists[t, EdgeWeight] is a useful analogue to ToAdjacencyLists when we are dealing with edge−weighted graphs and want the weights organized in a convenient fashion.

```
ToAdjacencyLists[SetEdgeWeights[t,
  WeightingFunction -> Random, WeightRange -> {2, 3}], EdgeWeight]
```

```
{{{1, 2.38054}, {2, 2.73307}, {9, 2.21706}, {9, 2.21706}, {10, 2.53599}},
 {{1, 2.73307}, {3, 2.9688}, {3, 2.9688}, {10, 2.85859}},
 {{2, 2.9688}, {2, 2.9688}, {4, 2.70583}, {10, 2.45127}},
 {{3, 2.70583}, {5, 2.29361}, {10, 2.84804}},
 {{4, 2.29361}, {6, 2.60012}, {10, 2.77135}},
 {{5, 2.60012}, {7, 2.30085}, {10, 2.48463}},
 {{6, 2.30085}, {8, 2.58848}, {10, 2.38553}},
 {{7, 2.58848}, {9, 2.13815}, {10, 2.01952}},
 {{1, 2.21706}, {1, 2.21706}, {8, 2.13815}, {10, 2.32987}},
 {{1, 2.53599}, {2, 2.85859}, {3, 2.45127}, {4, 2.84804},
  {5, 2.77135}, {6, 2.48463}, {7, 2.38553}, {8, 2.01952}, {9, 2.32987}}}
```

```
s = SetGraphOptions[t, EdgeDirection -> On];
```

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




```
- Graphics -
```

```
ToAdjacencyLists[s]
```

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

```
l3 = ToAdjacencyLists[t, Type -> Simple]
```

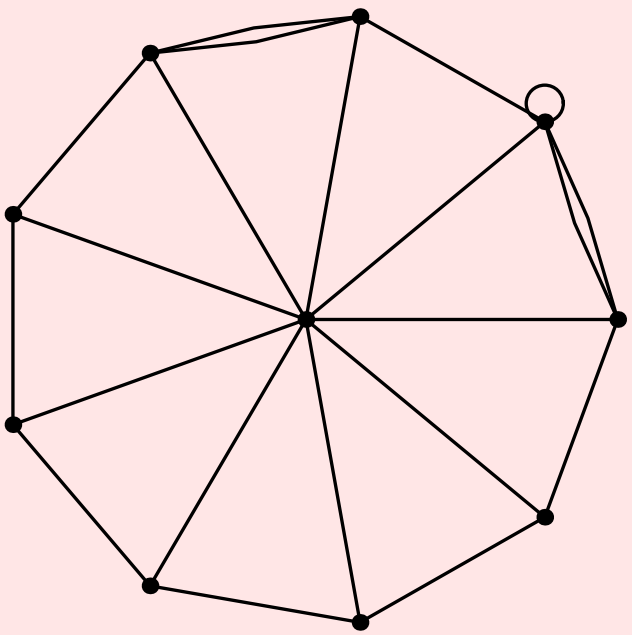
```
{{2, 9, 10}, {1, 3, 10}, {2, 4, 10}, {3, 5, 10}, {4, 6, 10}, {5, 7, 10},
 {6, 8, 10}, {7, 9, 10}, {1, 8, 10}, {1, 2, 3, 4, 5, 6, 7, 8, 9}}
```

FromAdjacencyLists

```
?FromAdjacencyLists
```

```
FromAdjacencyLists[l] constructs an edge list representation for
  a graph from the given adjacency lists l, using a circular
  embedding. FromAdjacencyLists[l, v] uses v as the embedding
  for the resulting graph. An option called Type that takes on
  the values Directed or Undirected, can be used to affect the
  type of graph produced. The default value of Type is Undirected.
```

```
ShowGraph[s = FromAdjacencyLists[l0, Vertices[t]]]
```



```
- Graphics -
```

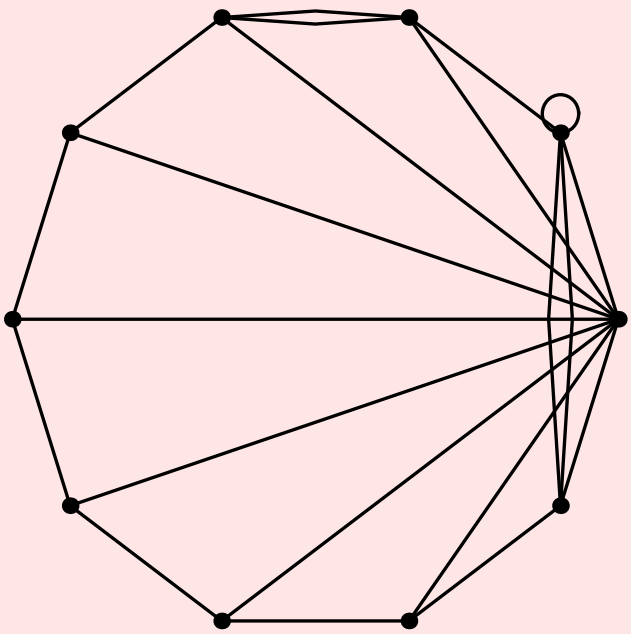
```
Edges[s]
```

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

NOTES

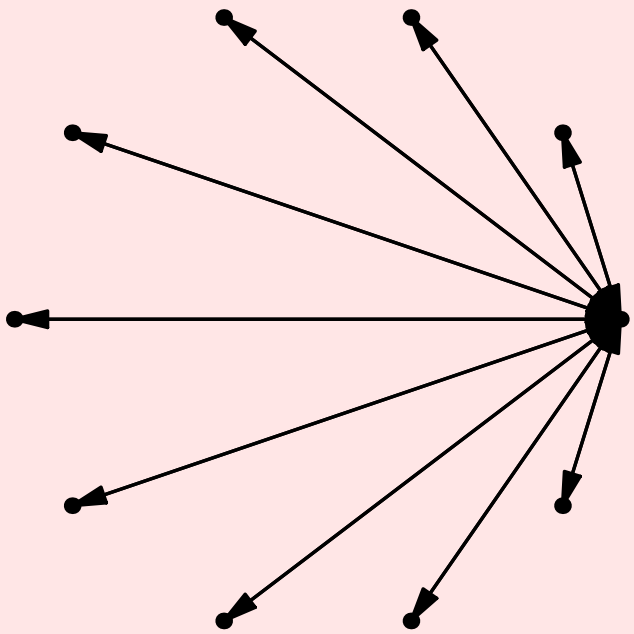
* Note that in going from edge list representation to adjacency list and back, multiple edges and self−loopsare preserved because these can be inferred from the adjacency list.

```
ShowGraph[FromAdjacencyLists[10]]
```



```
- Graphics -
```

```
ShowGraph[ FromAdjacencyLists[ ToAdjacencyLists[Star[10]], Type → Directed]]
```



```
- Graphics -
```

ToOrderedPairs

```
? ToOrderedPairs
```

```
ToOrderedPairs[g] constructs a list of ordered pairs representing the
  edges of the graph g. If g is undirected each edge is interpreted
  as two ordered pairs. An option called Type that takes on values
  Simple or All can be used to affect the constructed representation.
  Type -> Simple forces the removal of multiple edges and self-
  loops. Type -> All keeps all information and is the default option.
```

NOTES

* In going to ordered pairs, we view every undirected edge as a pair of directed edges. The option values for Type have obvious meanings. Note that in the example below since there are two undirected edges {1, 9}, these show up as two pairs {1, 9} and two pairs {9, 1}. The semantics of going from a directed graph to ordered pairs are obvious.

```
l0 = ToOrderedPairs[t]
```

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

```
l1 = ToOrderedPairs[t, Type → Simple]
```

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

```
s = SetGraphOptions[t, EdgeDirection → On];
```

```
ToOrderedPairs[ s ]
```

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

FromOrderedPairs

```
?FromOrderedPairs
```

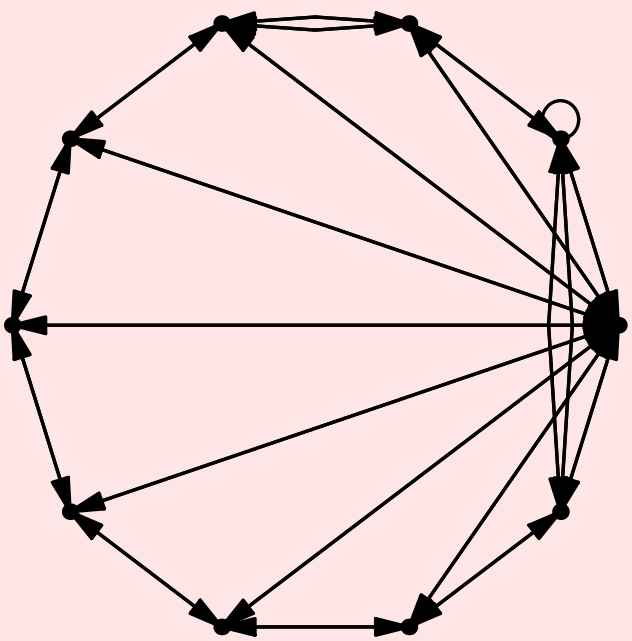
```
FromOrderedPairs[l] constructs an edge list representation from a list
  of ordered pairs l, using a circular embedding. FromOrderedPairs[l,
  v] uses v as the embedding for the resulting graph. The option Type
  that takes on values Undirected or Directed can be used to affect
  the kind of graph produced. The default value of Type is Directed.
  Type -> Undirected results in the underlying undirected graph.
```

NOTES

* The semantics of going from ordered pairs to a directed graph are obvious. In going from ordered pairs to an undirected graph, we get the underlying undirected graph. The Type –>Directed option is the default as well.

```
s = FromOrderedPairs[l0];
```

```
ShowGraph[s]
```



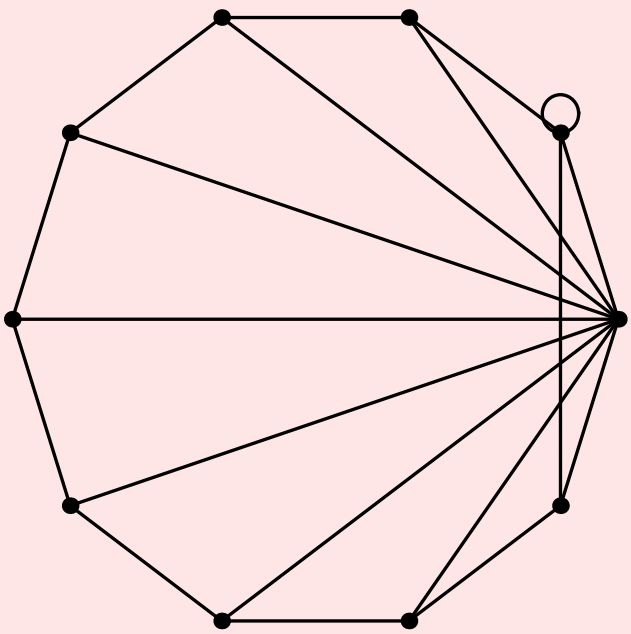
```
- Graphics -
```

```
s = FromOrderedPairs[l0, Type -> Undirected];
```

```
l0
```

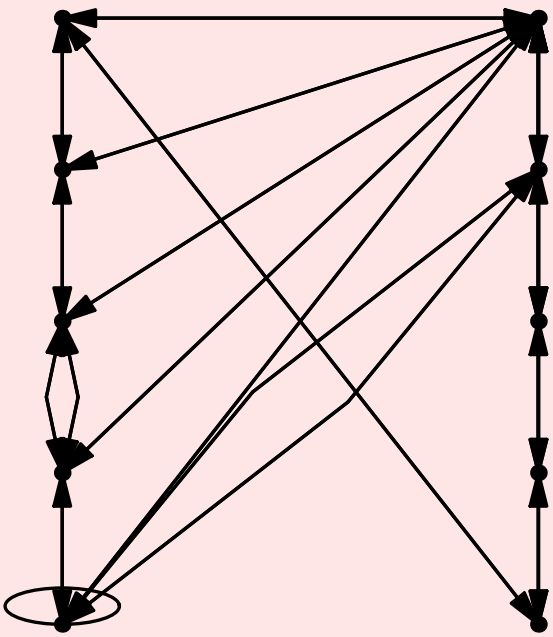
```
{{10, 1}, {10, 2}, {10, 3}, {10, 4}, {10, 5}, {10, 6}, {10, 7},
 {10, 8}, {10, 9}, {2, 1}, {3, 2}, {4, 3}, {5, 4}, {6, 5}, {7, 6}, {8, 7},
 {9, 8}, {9, 1}, {9, 1}, {3, 2}, {1, 10}, {2, 10}, {3, 10}, {4, 10},
 {5, 10}, {6, 10}, {7, 10}, {8, 10}, {9, 10}, {1, 2}, {2, 3}, {3, 4},
 {4, 5}, {5, 6}, {6,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

```
ShowGraph[s]
```



```
- Graphics -
```

```
ShowGraph[FromOrderedPairs[10, Vertices[CompleteGraph[5, 5]]]]
```



```
- Graphics -
```

BUG?

Why is the self−loopan ellipse? Somehow the aspect ratio became something other than 1 when the embedding is changed?

ToUnorderedPairs

```
?ToUnorderedPairs
```

```
ToUnorderedPairs[g] constructs a list of unordered pairs representing
  the edges of graph g. Each edge, directed or undirected, results in a
  pair in which the smaller vertex appears first. An option called Type
  that takes on values All or Simple can be used, and All is the default
  value. Type -> Simple ignores multiple edges and self-loops in g.
```

```
t
```

```
-Graph:<21, 10, Undirected>-
```

```
ToUnorderedPairs[t]
```

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

```
ToUnorderedPairs[t, Type -> NoSelfLoops]
```

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

```
ToUnorderedPairs[t, Type -> NoMultipleEdges]
```

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

```
ToUnorderedPairs[t, Type -> Simple]
```

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

```
s = SetGraphOptions[t, EdgeDirection -> On];
```

```
ToUnorderedPairs[ s ]
```

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

NOTES

* For both undirected and directed graphs in going to unordered pairs, the edges just get reported as they are.

* It is unclear if these functions to go back and forth between a graph and ordered/unordered pairs are needed any more, since the representation is essentially unordered/ordered pairs. I will check how often I use these functions; I seem to just use Edges[...] to get the edges of the graph.

FromUnorderedPairs

```
?FromUnorderedPairs
```

```
FromUnorderedPairs[l] constructs an edge list representation
  from a list of unordered pairs l, using a circular embedding.
  FromUnorderedPairs[l, v] uses v as the embedding for the
  resulting graph. The option Type that takes on values Undirected
  or Directed can be used to affect the kind of graph produced.
```

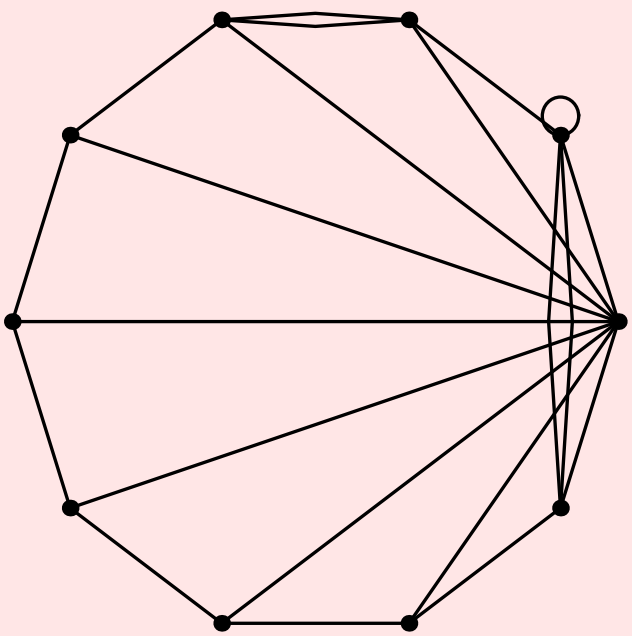
NOTES

* In going back from unordered pairs matters are straightforward since unordered pairs cannot be interpreted in different ways. The option Type –>Directed views every unordered pair as two directed edges.

```
l0 = ToUnorderedPairs[t]
```

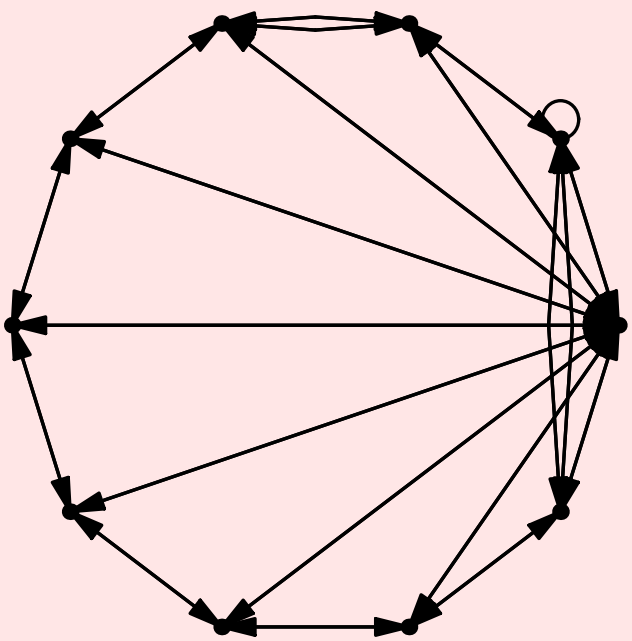
```
{{1, 10}, {2, 10}, {3, 10}, {4, 10}, {5, 10}, {6, 10},
 {7, 10}, {8, 10}, {9, 10}, {1, 2}, {2, 3}, {3, 4}, {4, 5},
 {5, 6}, {6, 7}, {7, 8}, {8, 9}, {1, 9}, {1, 1}, {1, 9}, {2, 3}}
```

```
ShowGraph[FromUnorderedPairs[l0]]
```



```
- Graphics -
```

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



```
- Graphics -
```

ToAdjacencyMatrix

```
?ToAdjacencyMatrix
```

```
ToAdjacencyMatrix[g] constructs an adjacency matrix representation
  for graph g. An option Type that takes on values All or Simple
  can be used to affect the graph constructed. Type -> All is the
  default, and Type -> Simple ignores any self-loops g may have.
  ToAdjacencyMatrix[g, EdgeWeight] returns edge weights as entries
  of the adjacency matrix with Infinity representing missing edges.
```

NOTES

* As things stand an adjacency matrix is assumed to be binary and hence cannot represent multiple edges, though it can represent self−loops.Should I change this to make an adjacency matrix be a non−negativeinteger matrix?

```
ToAdjacencyMatrix[t] // MatrixForm
```

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

```
ToAdjacencyMatrix[t, Type → Simple] // MatrixForm
```

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

```
k = SetEdgeWeights[RandomGraph[10, .3], WeightingFunction -> Euclidean];
```

```
ToAdjacencyMatrix[k, EdgeWeight] // TableForm
```

∞	∞	1.17557	∞	1.90211	∞
∞	∞	∞	∞	1.61803	1.90211
1.17557	∞	∞	0.618034	∞	∞
∞	∞	0.618034	∞	∞	∞
1.90211	1.61803	∞	∞	∞	∞
∞	1.90211	∞	∞	∞	∞
1.90211	2.	∞	∞	1.17557	∞
1.61803	∞	∞	∞	1.61803	∞
1.17557	∞	∞	∞	∞	1.61803
∞	1.17557	∞	1.90211	∞	∞

```
g = GraphUnion[ CompleteGraph[2], CompleteGraph[2], CompleteGraph[2]];
```

```
ToAdjacencyMatrix[g, EdgeWeight]
```

```
{{∞, 1, ∞, ∞, ∞, ∞}, {1, ∞, ∞, ∞, ∞, ∞}, {∞, ∞, ∞, 1, ∞, ∞},
 {∞, ∞, 1, ∞, ∞, ∞}, {∞, ∞, ∞, ∞, ∞, 1}, {∞, ∞, ∞, ∞, 1, ∞}}
```

```
ToAdjacencyMatrix[g] // MatrixForm
```

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

FromAdjacencyMatrix

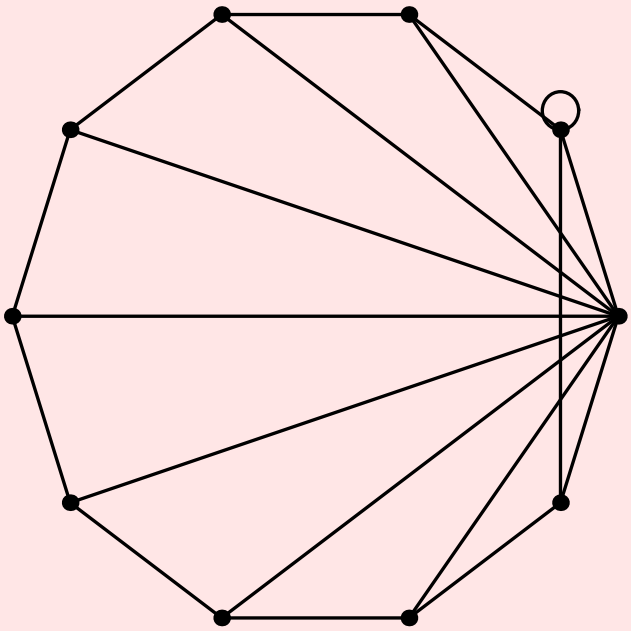
```
?FromAdjacencyMatrix
```

```
FromAdjacencyMatrix[m] constructs an edge list representation
  for a graph with adjacency matrix m, using a circular
  embedding. FromAdjacencyMatrix[m, v] uses v as the embedding
  for the resulting graph. An option Type that takes on the
  values Directed or Undirected can be used to affect the type
  of graph produced. The default value of Type is Undirected.
```

```
tg = FromAdjacencyMatrix[ToAdjacencyMatrix[t]]
```

```
-Graph:<19, 10, Undirected>-
```

```
ShowGraph[tg]
```



```
- Graphics -
```

PseudographQ

```
?PseudographQ
```

```
PseudographQ[g] yields True if graph
  g is a pseudograph, meaning it contains self-loops.
```

```
PseudographQ[t]
```

```
True
```

```
PseudographQ[DeleteEdges[t, {{1, 1}}]]
```

```
False
```

UnweightedQ

```
? UnweightedQ
```

```
UnweightedQ[g] yields True if all edge weights are 1 and False otherwise.
```

```
UnweightedQ[t]
```

```
True
```

```
UnweightedQ[ SetEdgeWeights[t, WeightingFunction → Random]]
```

```
False
```

SimpleQ

```
? SimpleQ
```

```
SimpleQ[g] yields True if g is a simple graph,
  meaning it has no multiple edges and contains no self-loops.
```

```
SimpleQ[t]
```

```
False
```

```
s = RemoveSelfLoops[t]; SimpleQ[s]
```

```
False
```

```
s = RemoveMultipleEdges[s]; SimpleQ[s]
```

```
True
```

RemoveSelfLoops

```
?RemoveSelfLoops
```

```
RemoveSelfLoops[g] returns the
  graph obtained by deleting self-loops in g.
```

```
s = AddEdges[t, {{{2, 2}}}];
```

```
PseudographQ[s]
```

```
True
```

```
PseudographQ[RemoveSelfLoops[s]]
```

```
False
```

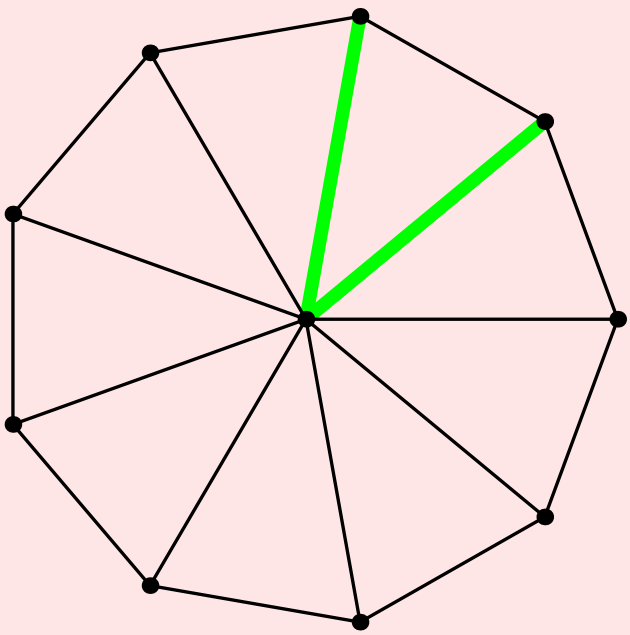
RemoveMultipleEdges

```
?RemoveMultipleEdges
```

```
RemoveMultipleEdges[g] returns the
  graph obtained by deleting multiple edges from g.
```

```
s = SetGraphOptions[Wheel[10],
    {{{1, 10}, {2, 10}, EdgeStyle -> Fat, EdgeColor -> Green}}];
```

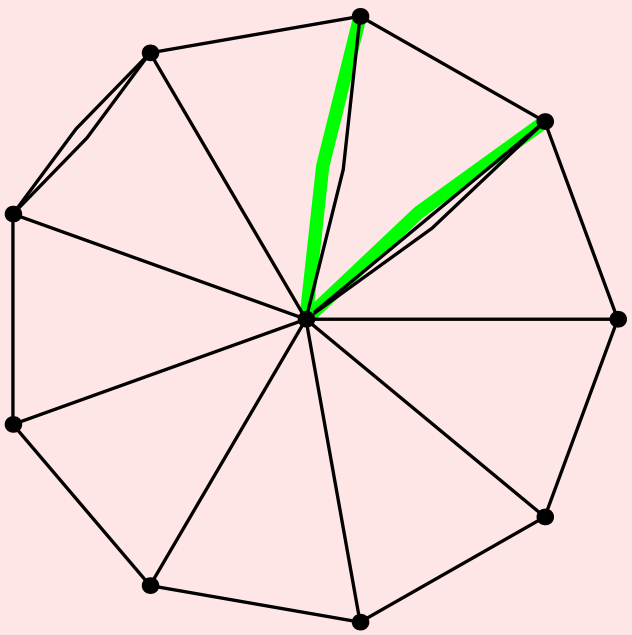
```
ShowGraph[ s ]
```



```
- Graphics -
```

```
s = AddEdges[s, {{{1, 10}}, {{1, 10}}, {{2, 10}}, {{3, 4}}}];
```

```
ShowGraph[ s ]
```



```
- Graphics -
```

```
M[s]
```

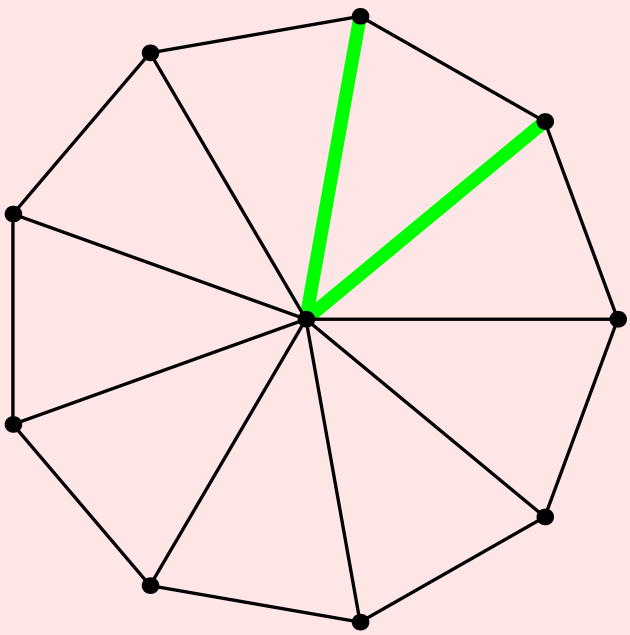
```
22
```

```
ss = RemoveMultipleEdges[s];
```

```
M[ss]
```

```
18
```

```
ShowGraph[ss]
```



```
- Graphics -
```

NOTES

* Suppose there are three edges {a, b} in a graph. RemoveMultipleEdges removes two copies of {a, b}. The choice of which two to remove is arbitrary and the user cannot depend on which two edges the function removes. This is especially so because no assumption can be made about the order in which edges are listed in the edge–listrepresentation.

EmptyQ

```
?EmptyQ
```

```
EmptyQ[g] yields True if graph g contains no edges.
```

```
EmptyQ[DeleteEdges[CompleteGraph[2], {{1, 2}}]]
```

```
True
```

```
EmptyQ[t]
```

```
False
```

CompleteQ

```
?CompleteQ
```

```
CompleteQ[g] yields True if graph g is complete.  This means
  that between any pair of vertices there is an undirected
  edge or two directed edges going in opposite directions.
```

```
CompleteQ[t]
```

```
False
```

```
CompleteQ[CompleteGraph[10]]
```

```
True
```

```
CompleteQ[s = DeleteEdges[CompleteGraph[3], {{1, 2}}]]
```

```
False
```

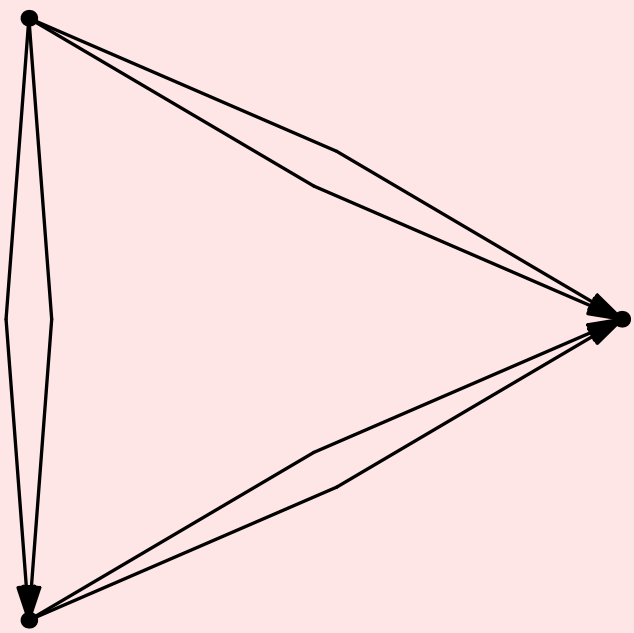
```
s = AddEdges[EmptyGraph[3],
   {{{1, 2}}, {{1, 3}}, {{2, 3}}, {{2, 1}}, {{3, 1}}, {{3, 2}}}];
```

```
CompleteQ[s]
```

```
False
```

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

```
ShowGraph[s]
```



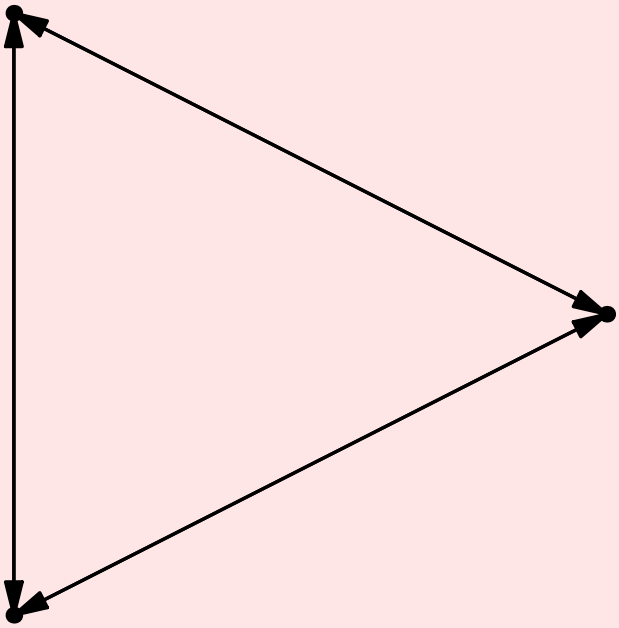
```
- Graphics -
```

```
CompleteQ[s]
```

```
False
```

```
s = AddEdges[SetGraphOptions[EmptyGraph[3], EdgeDirection → On],
   {{{1, 2}}, {{1, 3}}, {{2, 3}}, {{2, 1}}, {{3, 1}}, {{3, 2}}}];
```

```
ShowGraph[s]
```



```
- Graphics -
```

```
CompleteQ[s]
```

```
True
```

MultipleEdgesQ

```
?MultipleEdgesQ
```

```
MultipleEdgesQ[g] yields True if g has multiple
  edges between pairs of vertices. It yields False otherwise.
```

```
Sort[Edges[t]]
```

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

```
MultipleEdgesQ[t]
```

```
True
```

```
MultipleEdgesQ[DeleteEdges[t, {{2, 3}, {1, 9}}]]
```

```
False
```

```
s = AddEdges[t, {{{2, 3}}}]; MultipleEdgesQ[ DeleteEdges[s, {{2, 3}, {1, 9}}]]
```

```
True
```

```
MultipleEdgesQ[ DeleteEdges[s, {{2, 3}, {1, 9}}, All]]
```

```
False
```

InduceSubgraph

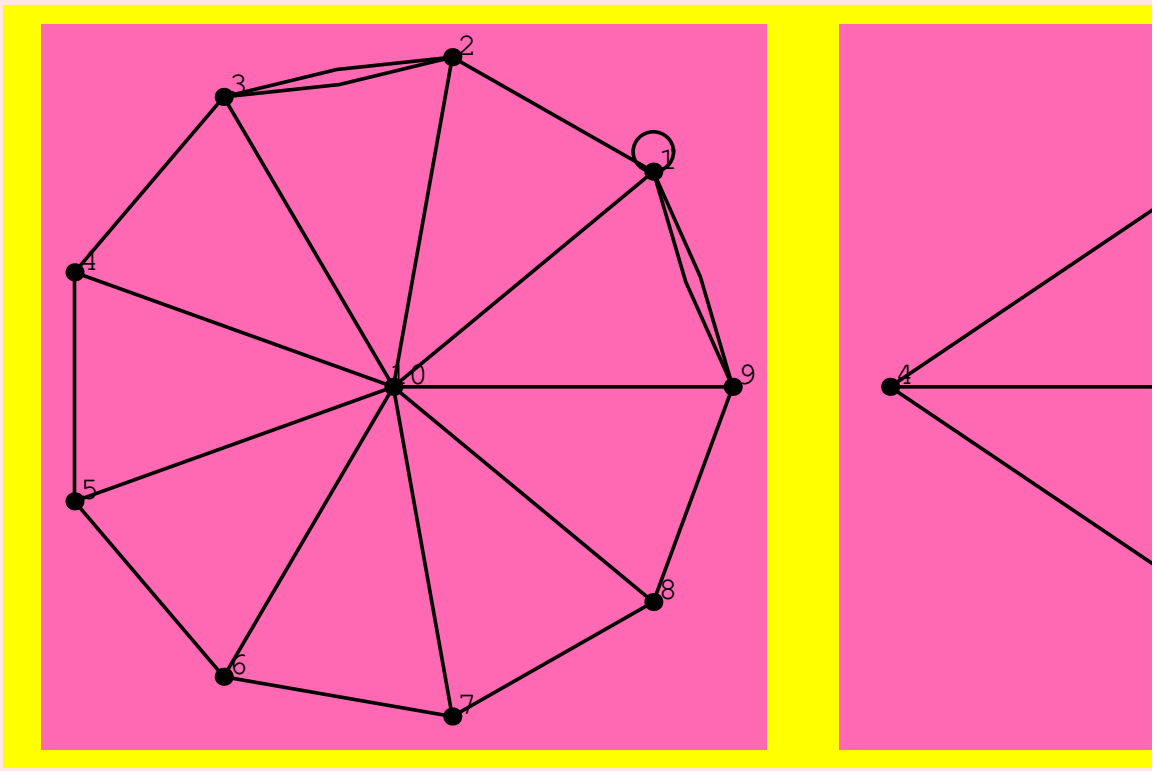
```
? InduceSubgraph
```

```
InduceSubgraph[g, s] constructs the
  subgraph of graph g induced by the list of vertices s.
```

```
p1 = ShowGraph[t, Background -> HotPink,
  VertexNumber -> Text[{0.02, 0.02}, TextStyle -> {FontSize -> 12}], Graphics];
p2 = ShowGraph[InduceSubgraph[t, {1, 9, 8, 10}], Background -> HotPink,
  VertexNumber -> Text[{0.02, 0.02}, TextStyle -> {FontSize -> 12}], Graphics]
```

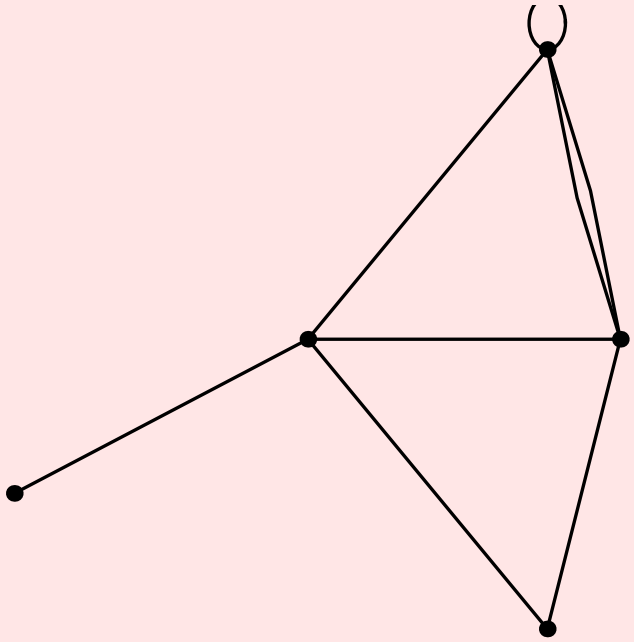
```
- Graphics -
```

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



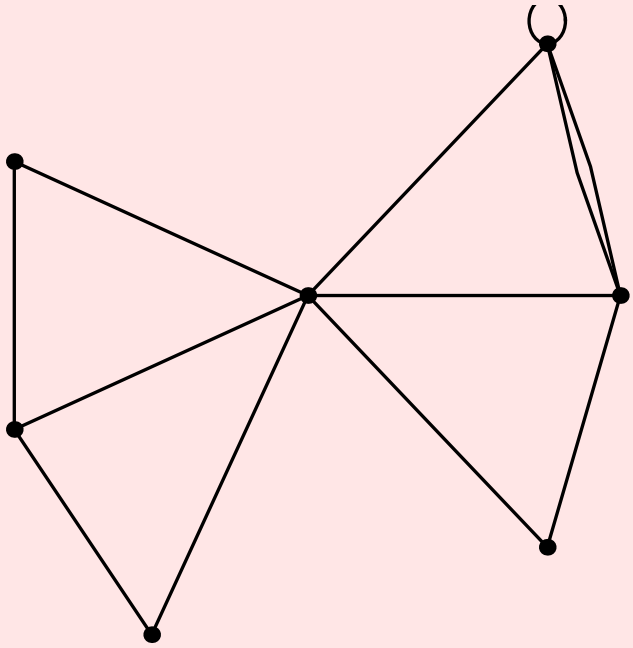

```
- GraphicsArray -
```

```
ShowGraph[InduceSubgraph[t, {1, 9, 8, 10, 5}]]
```



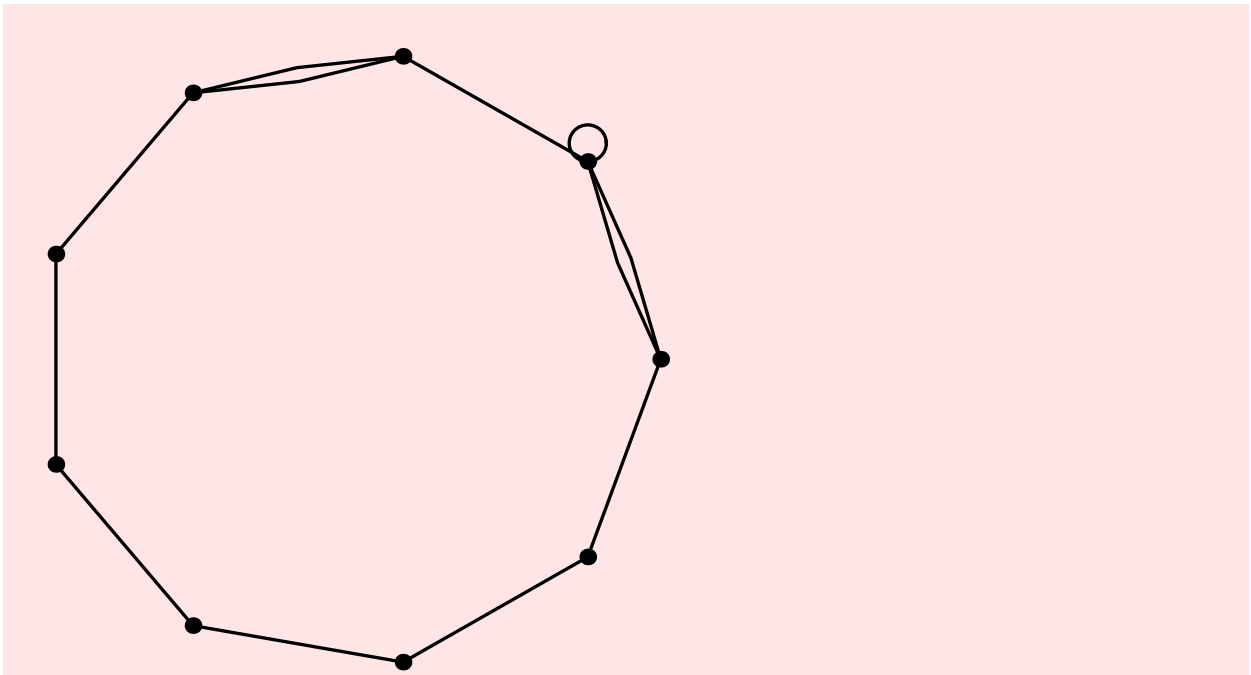
```
- Graphics -
```

```
ShowGraph[InduceSubgraph[t, {1, 9, 8, 10, 5, 4, 6}]]
```



```
- Graphics -
```

```
ShowGraph[InduceSubgraph[t, {1, 2, 4, 3,  5, 6, 7, 8, 9}]]
```



```
- Graphics -
```

NOTES

* Note that the list of vertices need be specified in order and may contain duplicates.

* Below I compare the times of InduceSubgraph in the old and the new versions. It is hard to beat the old implementation because InduceSubgraph is so naturally and simply expressed as simple matrix operations on the adjacency matrix.

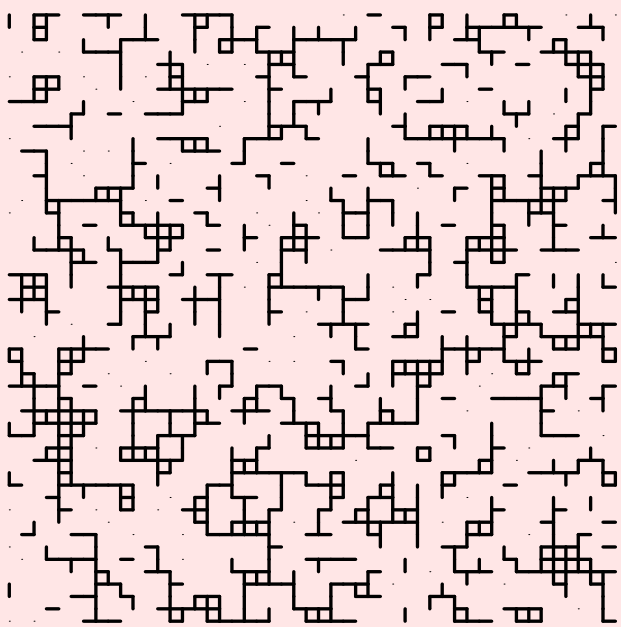
```
g1 = DiscreteMath`OldCombinatorica`GridGraph[30, 30]; g2 = GridGraph[30, 30];
```

```
s = RandomSubset[Range[900]];
```

```
{Timing[DiscreteMath`OldCombinatorica`InduceSubgraph[g1, s];],
 Timing[InduceSubgraph[g2, s];]}
```

```
{{0.156 Second, Null}, {0.234 Second, Null}}
```

```
ShowGraph[InduceSubgraph[GridGraph[50, 50], RandomSubset[Range[2500]]],
 VertexStyle -> Disc[0]]
```



```
- Graphics -
```

NOTES

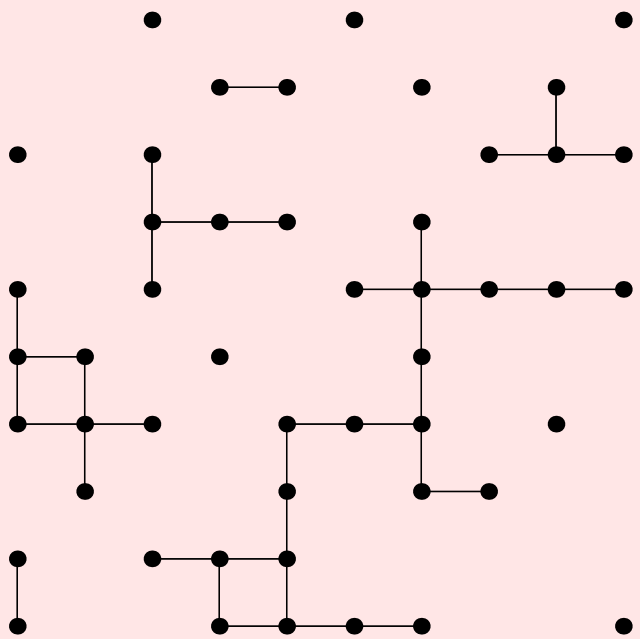
* One of the CUP referees had complained that InduceSubgraph does not preserve original embedding for the subgraph. This is not quite true as the following examples show. However, InduceSubgraph is clever in the sense that the subset of vertices is also viewed as a permutation and the vertex embedding is permuted accordingly. While this is a natural consequence of the implementation of InduceSubgraph, I agree with the referee that viewing subsets as permutations also in this context is a bit of overload.

* So I have separated the two tasks into two functions: InduceSubgraph and PermuteSubgraph. InduceSubgraph simply calls PermuteSubgraph with a sorted subset. As a consequence, instead of InduceSubgraph, we use PermuteSubgraph in functions related to graph isomorphism. PermuteSubgraph has the advantage of providing new embeddings as a result of permuting the vertices, something that InduceSubgraph did not do.

```
r = RandomSubset[Range[100]];
```

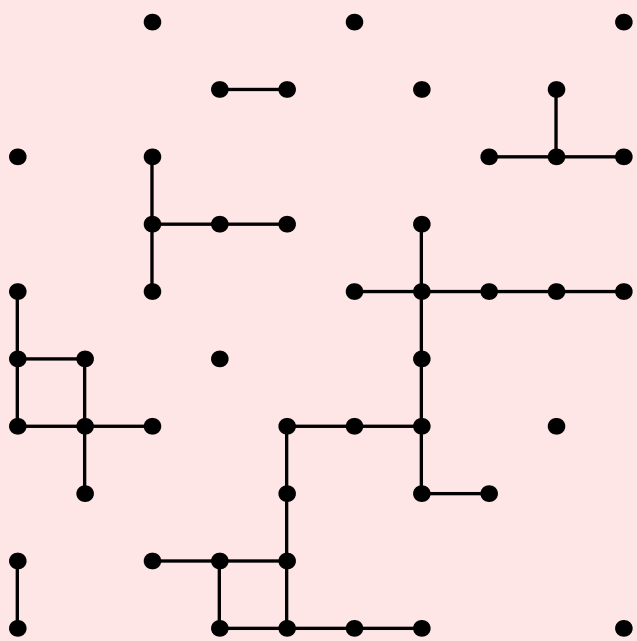
```
rp = Permute[r, RandomPermutation[Length[r]]];
```

```
DiscreteMath`OldCombinatorica`ShowGraph[
 DiscreteMath`OldCombinatorica`InduceSubgraph[
  DiscreteMath`OldCombinatorica`GridGraph[10, 10], r]]
```



```
- Graphics -
```

```
ShowGraph[InduceSubgraph[GridGraph[10, 10], rp]]
```



```
- Graphics -
```

PermuteSubgraph

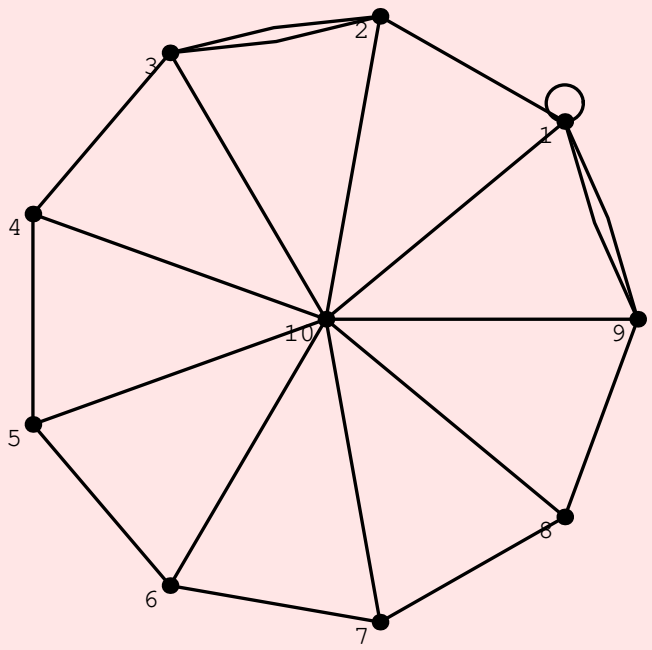
```
?PermuteSubgraph
```

```
PermuteSubgraph[g, p] permutes the vertices
  of a subgraph of g induced by p according to p.
```

NOTES

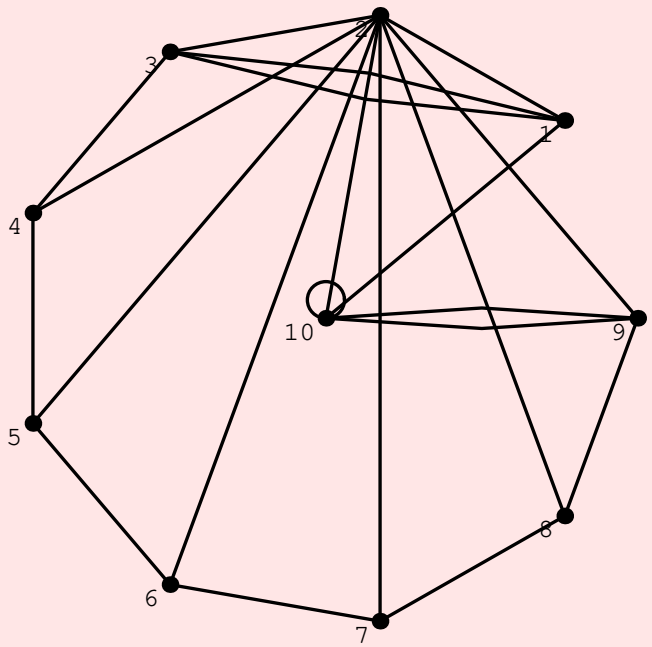
* In the example below, 1 –>2, 2 –>10, and 10–>1 with everything else staying in place.

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



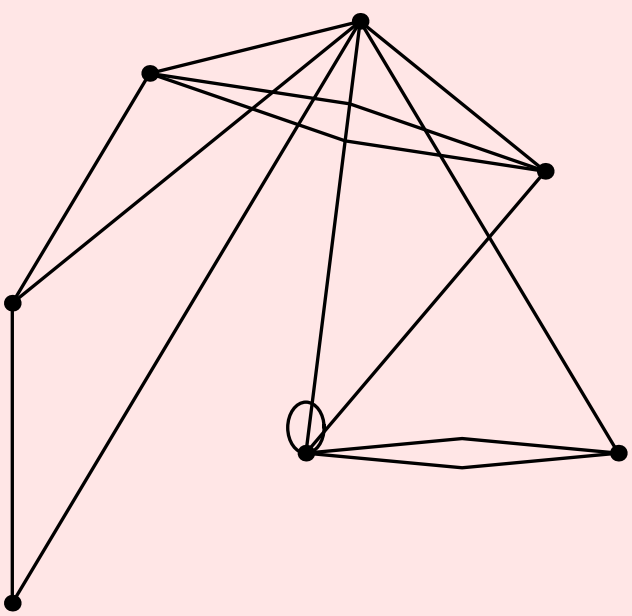

```
- Graphics -
```

```
ShowGraph[PermuteSubgraph[t, {2, 10, 3, 4, 5, 6, 7, 8, 9, 1}],
 VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[PermuteSubgraph[t, {2, 10, 3, 4, 5,  9, 1}]]
```

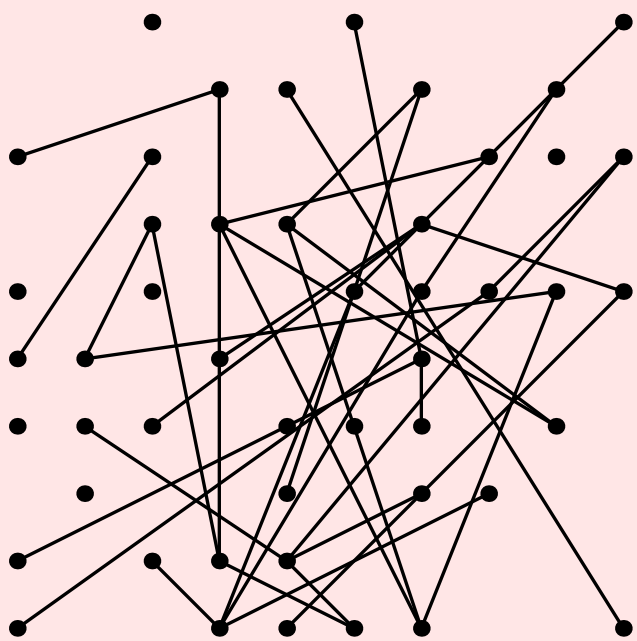


```
- Graphics -
```

```
Length[rp]
```

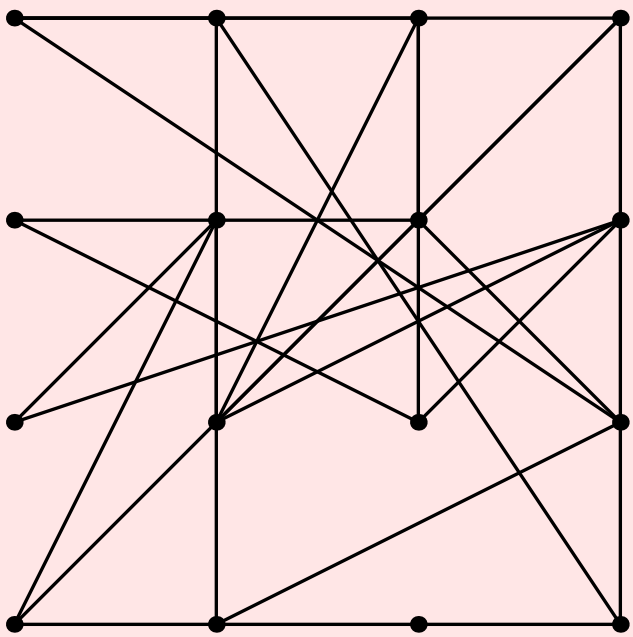
```
48
```

```
ShowGraph[PermuteSubgraph[GridGraph[10, 10], rp]]
```



```
- Graphics -
```

```
ShowGraph[ PermuteSubgraph[ GridGraph[4, 4], RandomPermutation[16] ] ]
```



```
- Graphics -
```

NOTES

* This is another way to get new and possibly interesting embeddings of graphs.

* Note how this differs from InduceSubgraph.

Contract

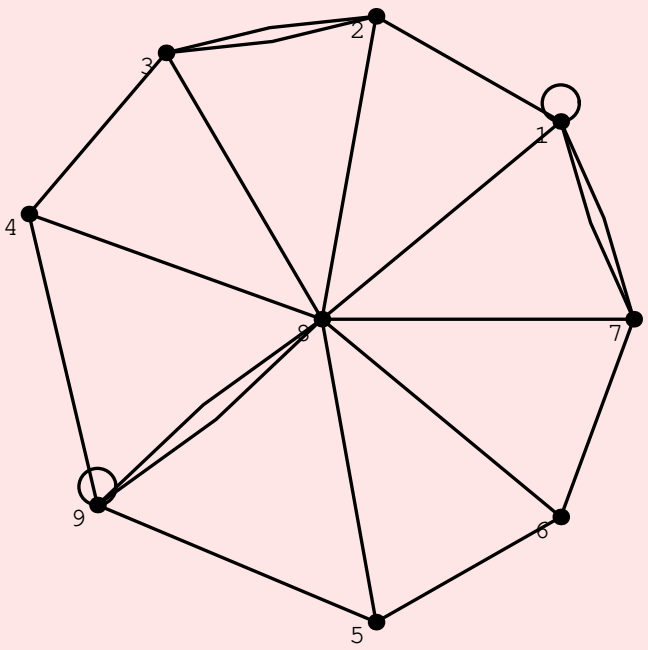
```
?Contract
```

```
Contract[g, {x, y}] gives the graph resulting
  from contracting the pair of vertices {x, y} of graph g.
```

NOTES

* In Steve Skiena's description of Contract in the book, it talks about contracting an edge. I don't see why contract should not be defined for ANY pair of vertices. Also, when a pair of vertices that have an edge connecting them is contracted, the edge becomes a self−loop.This is the definition of contract in West's graph theory book.

```
ShowGraph[s = Contract[t, {5, 6}], VertexNumber -> On]
```

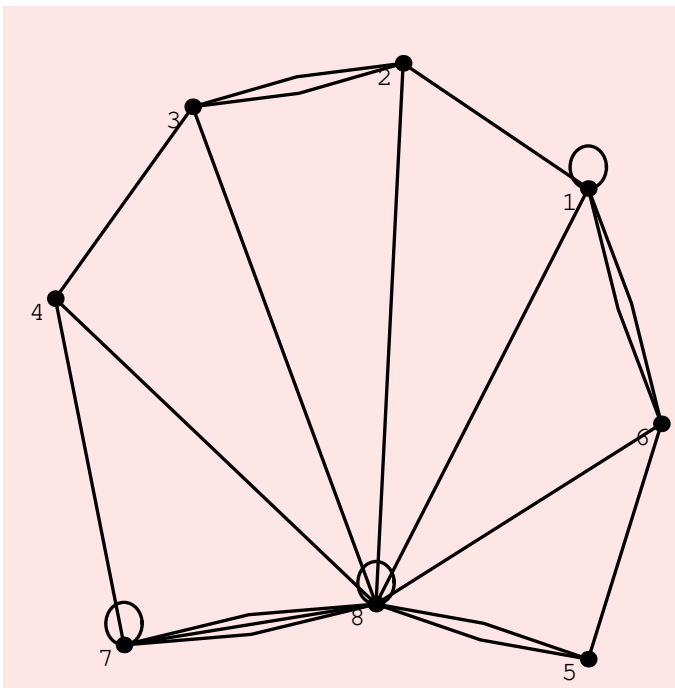



```
- Graphics -
```

NOTES

* Note the self−loopsin s. When 5 and 6 were contracted into vertex 9 we got a self−loop{9. 9}. Also note the multiple edges {9, 8} −since both 5 and 6 have edges to 8, we get the multiple edges {9, 8}.

```
ShowGraph[s = Contract[s, {5, 8}], VertexNumber -> On]
```




```
- Graphics -
```

GraphComplement

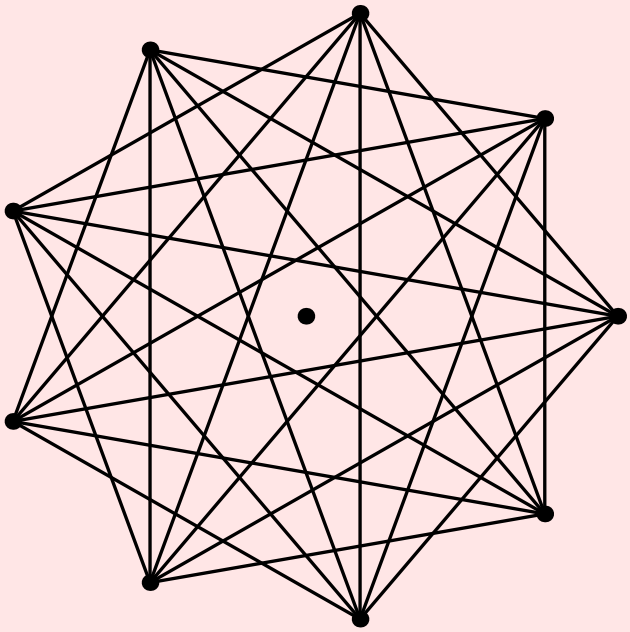
```
? GraphComplement
```

```
GraphComplement[g] gives the complement of graph g.
```

```
s = Graph[ {}, {}]; GraphComplement[s]
```

```
-Graph:<0, 0, Undirected>-
```

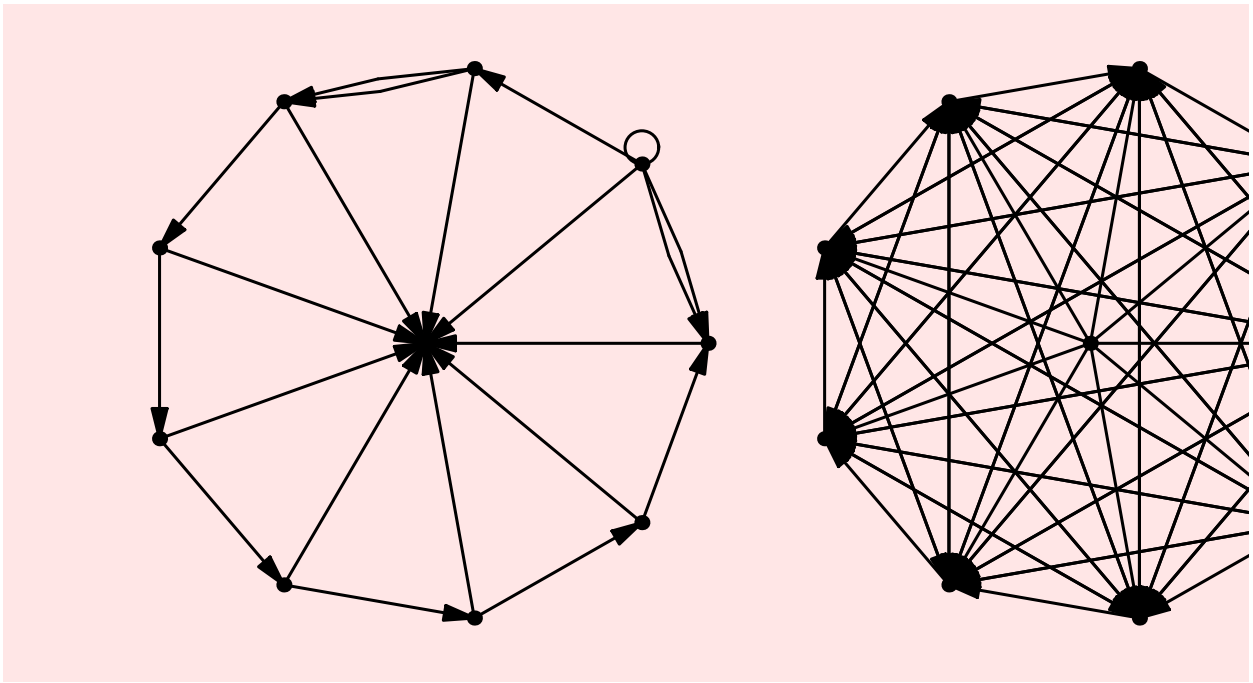
```
ShowGraph[GraphComplement[t]]
```



```
- Graphics -
```

```
p1 = ShowGraph[s = SetGraphOptions[t, EdgeDirection -> On], Graphics];
p2 = ShowGraph[GraphComplement[s], Graphics];
```

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

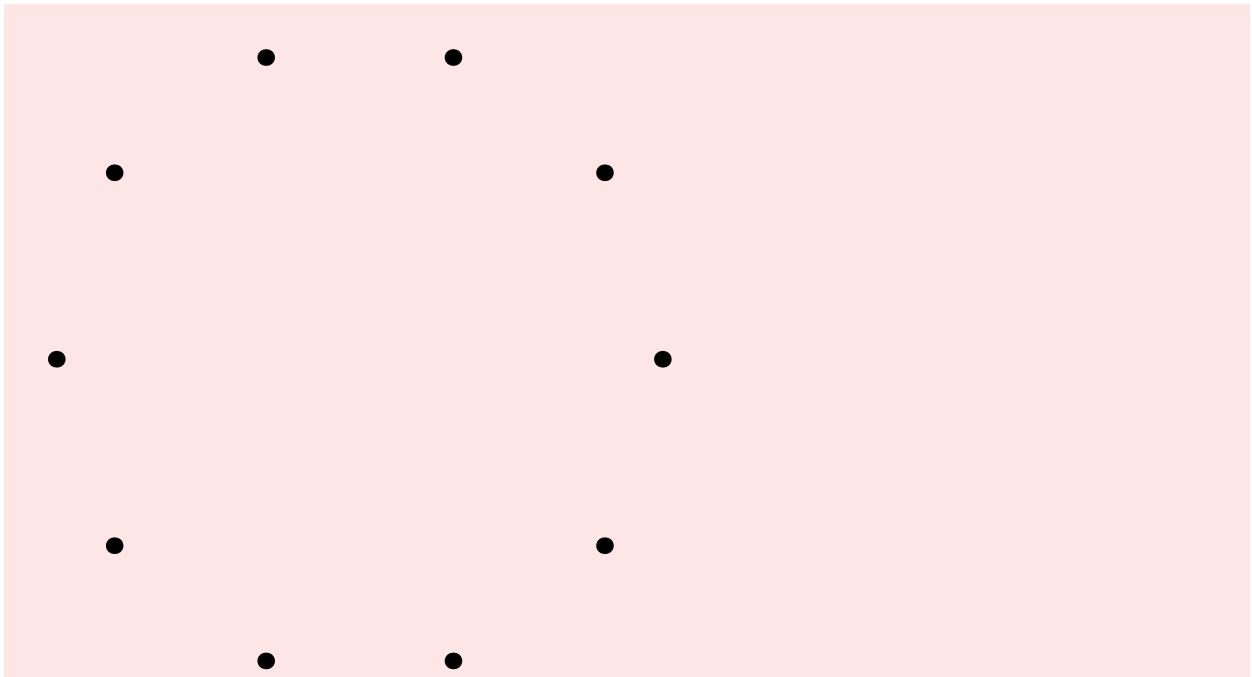


```
- GraphicsArray -
```

NOTES

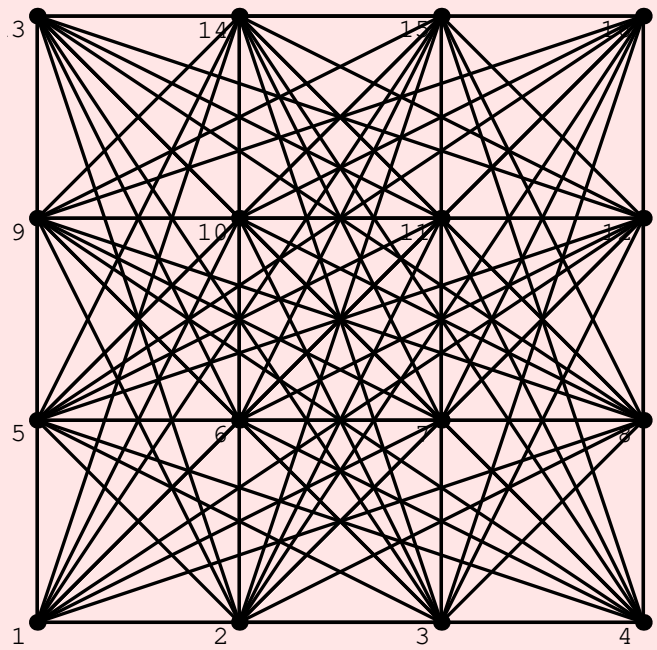
* As can be seen from the above examples, GraphComplement works as expected for undirected as well as for directed graphs.

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



```
- Graphics -
```

```
ShowGraph[s = GraphComplement[ GridGraph[4, 4]], VertexNumber -> On]
```




```
- Graphics -
```

MakeUndirected

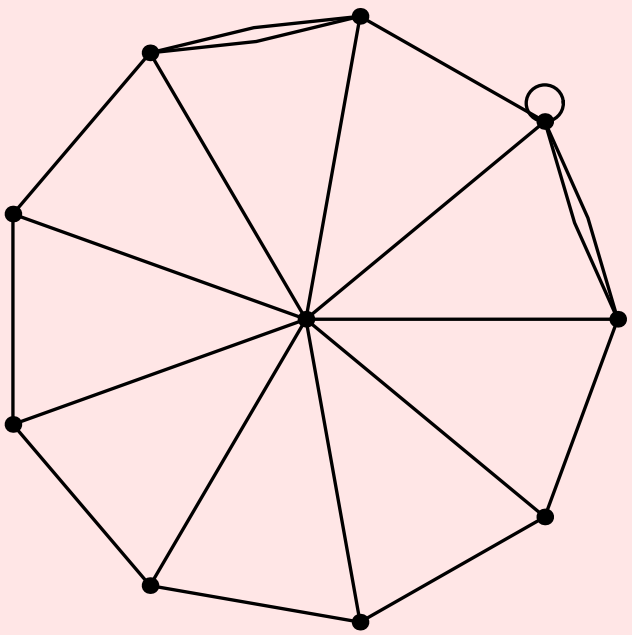
```
?MakeUndirected
```

```
MakeUndirected[g] gives the underlying
  undirected graph of the given directed graph g.
```

```
s = MakeUndirected[t]
```

```
-Graph:<21, 10, Undirected>-
```

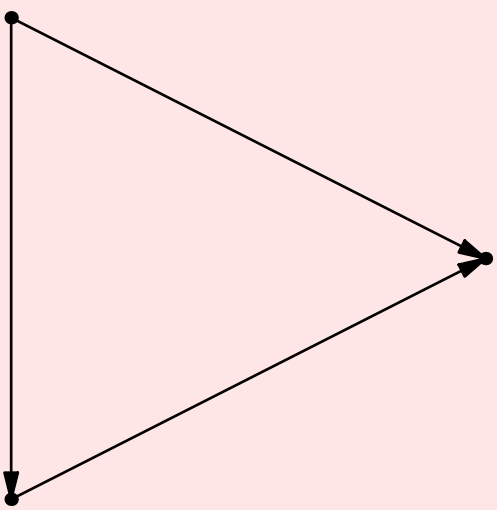
```
ShowGraph[s]
```



```
- Graphics -
```

```
s = SetGraphOptions[CompleteGraph[3], EdgeDirection -> On];
```

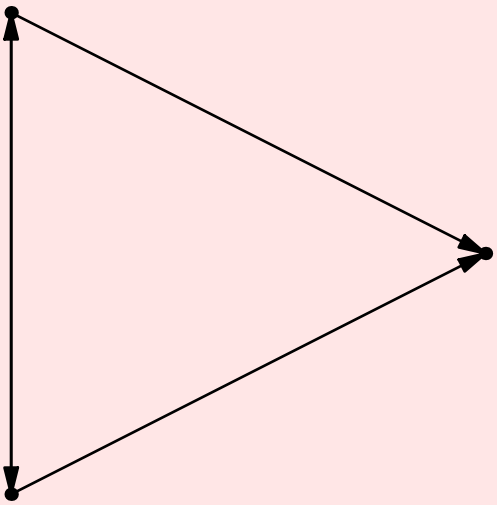
```
ShowGraph[s, ImageSize -> 200]
```



```
- Graphics -
```

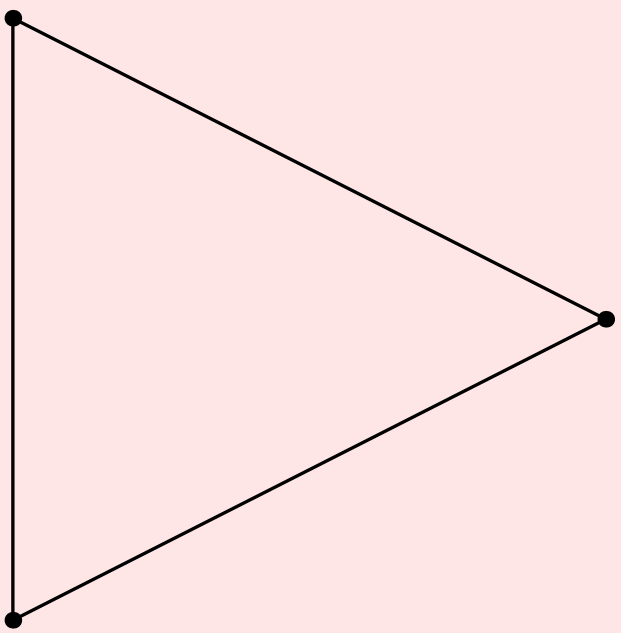
```
s = AddEdges[s, {{{2, 1}}}];
```

```
ShowGraph[s, ImageSize -> 200]
```



```
- Graphics -
```

```
ShowGraph[MakeUndirected[s]]
```



```
- Graphics -
```

```
SetGraphOptions[s, EdgeDirection -> Off]
```

```
-Graph:<4, 3, Undirected>-
```

NOTES

* Note that MakeUndirected takes two directed edges going in opposite directions and converts them into two undirected edges. Also note the difference between MakeUndirected and simply setting the EdgeDirection Off. MakeUndirected also makes sure that the edges go from lower numbered vertices to higher numbered vertices. In other words, it is a bad idea to turn edge–directionsoff by using SetGraphOptions[...].

UndirectedQ

```
?UndirectedQ
```

```
UndirectedQ[g] yields True if graph g is undirected.
```

```
UndirectedQ[t]
```

```
True
```

```
UndirectedQ[s = SetGraphOptions[t, EdgeDirection -> On]]
```

```
False
```

```
UndirectedQ[MakeUndirected[s = SetGraphOptions[t, EdgeDirection -> On]]]
```

```
True
```

MakeSimple

```
?MakeSimple
```

```
MakeSimple[g] gives the undirected graph, free
  of multiple edges and self-loops derived from graph g.
```

```
tt = MakeSimple[t];
```

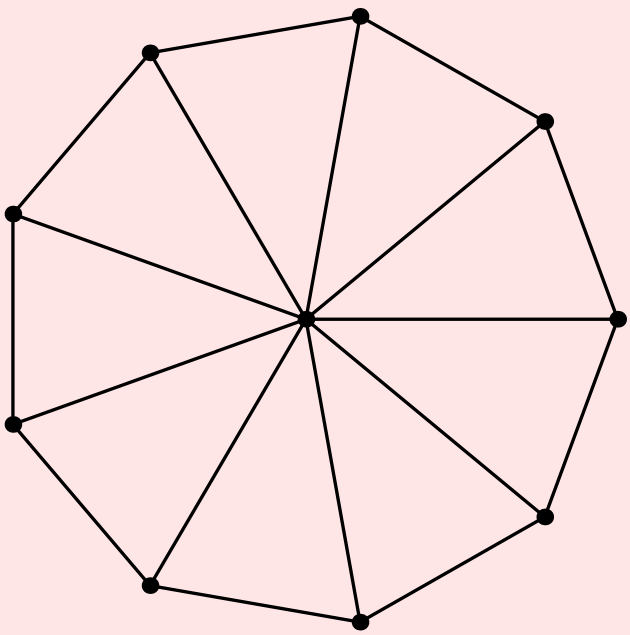
```
M[tt]
```

```
18
```

```
SimpleQ[tt]
```

```
True
```

```
ShowGraph[ tt ]
```



```
- Graphics -
```

DepthFirstTraversal

```
?DepthFirstTraversal
```

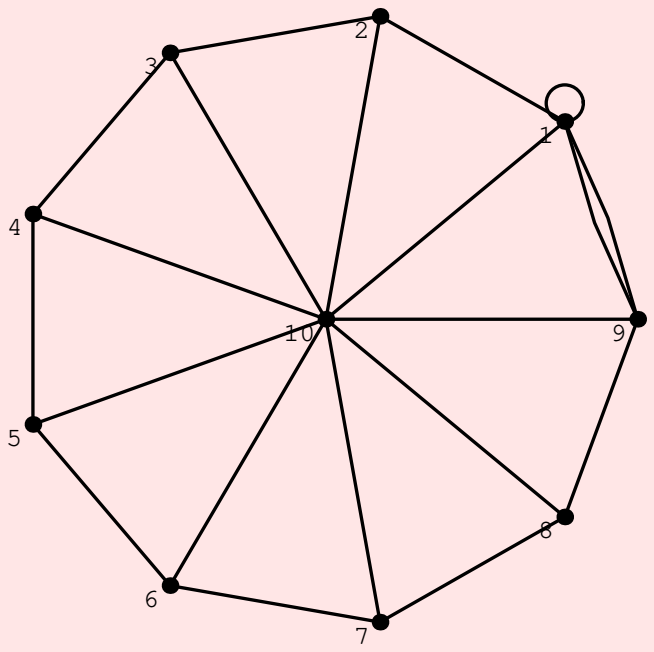
```
DepthFirstTraversal[g, v] performs a depth-first traversal of graph
  g starting from vertex v, and gives a list of vertices in the
  order in which they were encountered. DepthFirstTraversal[g, v,
  Edge] returns the edges of the graph that are traversed by the
  depth-first traversal in the order in which they are traversed.
```

TO DO

* DFS should be made more powerful and flexible –it should in addition return a partition of the edges into tree edges, back edges, forward edges, and cross edges. This should be done through various options to DFS.

```
s = DeleteEdges[t, {{2, 3}}];
```

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



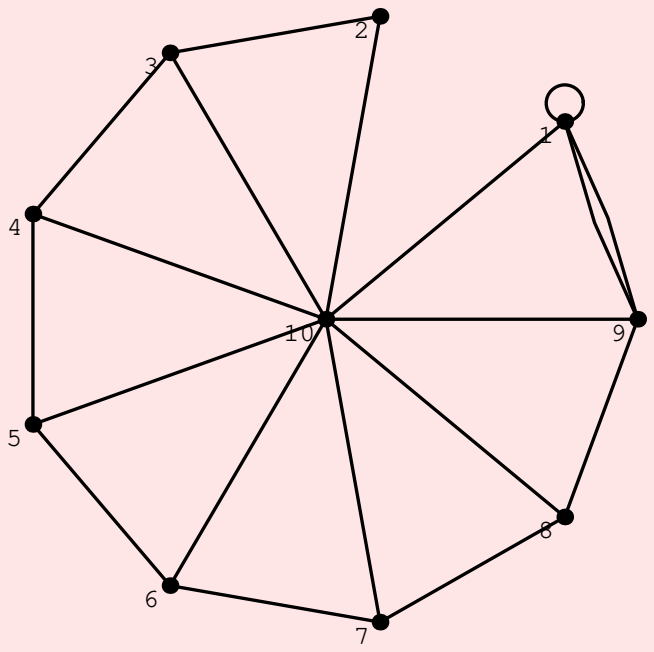

```
- Graphics -
```

```
DepthFirstTraversal[s, 2]
```

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

```
s = DeleteEdges[s, {{1, 2}}];
```

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



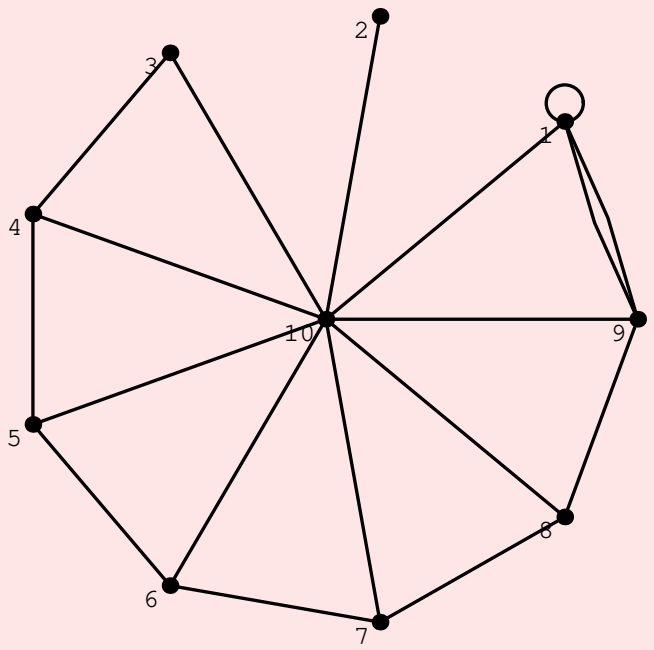

```
- Graphics -
```

```
DepthFirstTraversal[s, 2]
```

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

```
s = DeleteEdges[s, {{2, 3}}];
```

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

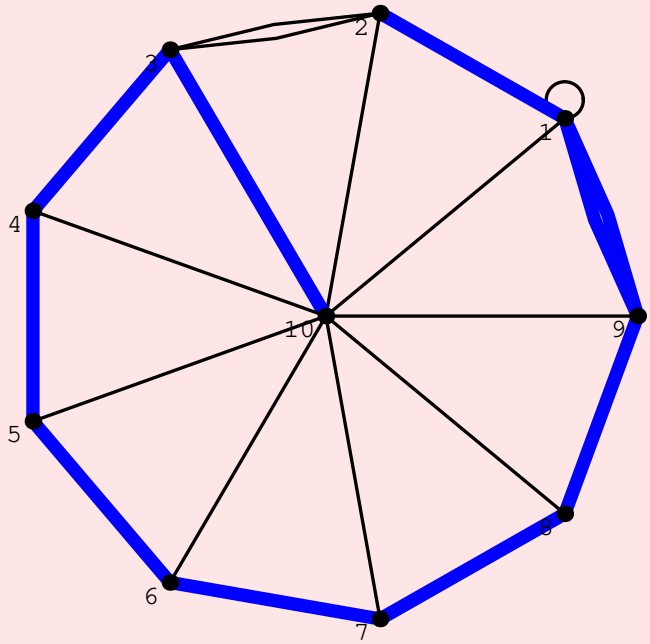



```
- Graphics -
```

```
DepthFirstTraversal[s, 2]
```

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

```
ShowGraph[Highlight[t, {Map[Sort, DepthFirstTraversal[t, 2, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexNumber -> On]
```




```
- Graphics -
```

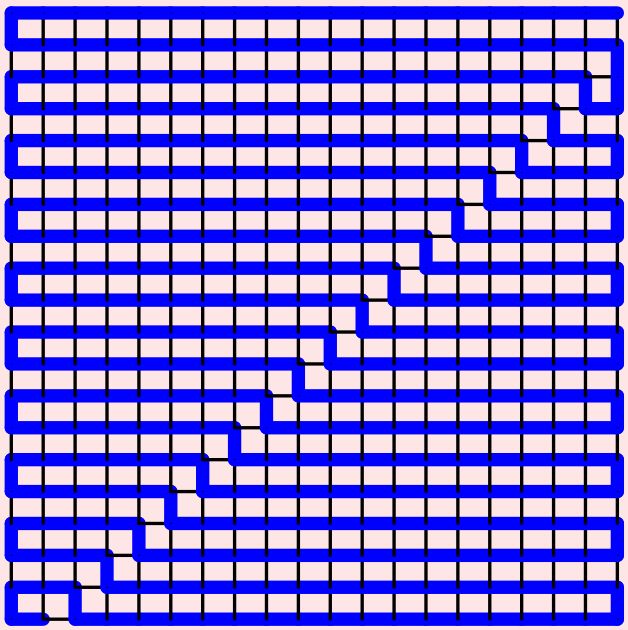
NOTES

* Note that multiple edges are highlighted correctly –and only one of the edges {1,9} is highlighted.

```
$RecursionLimit = 20000;
```

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

```
ShowGraph[Highlight[s, {Map[Sort, DepthFirstTraversal[s, 2, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexStyle -> Disc[0]]
```



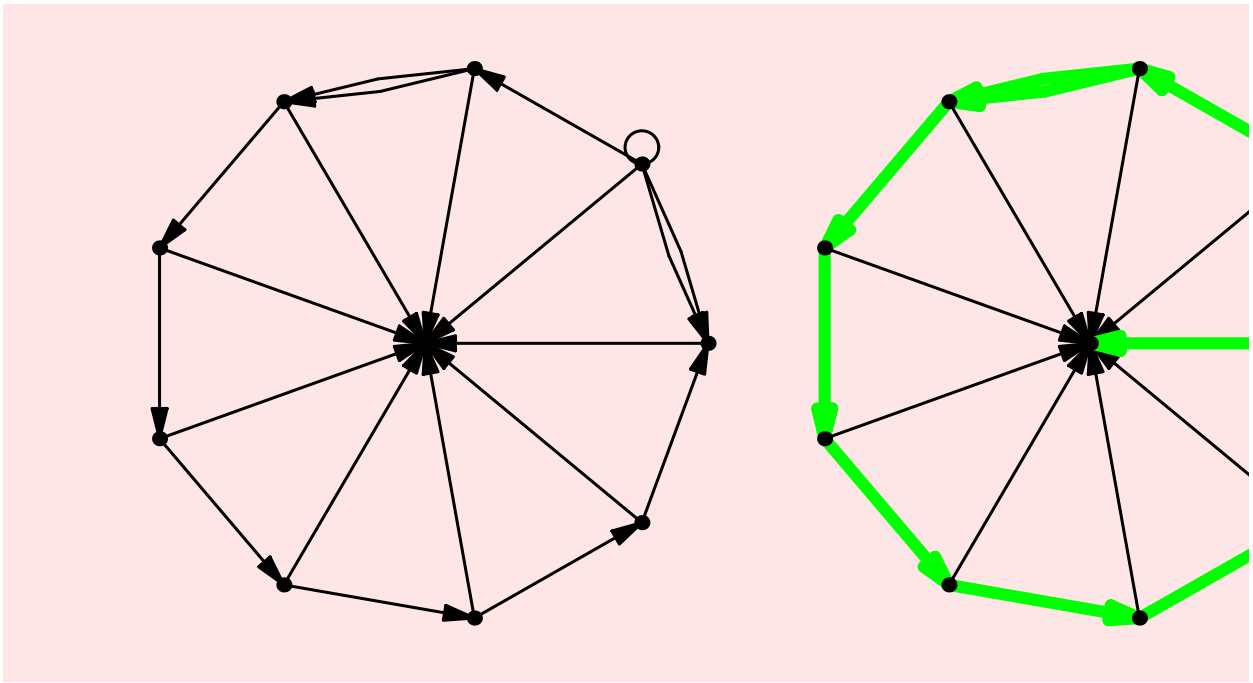
```
- Graphics -
```

```
DepthFirstTraversal[s = SetGraphOptions[t, EdgeDirection -> On], 1, Edge]
```

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

```
p1 = ShowGraph[s, Graphics];
p2 = ShowGraph[ Highlight[s, {DepthFirstTraversal[s , 1, Edge]},
   HighlightedEdgeColors -> {Green}], Graphics];
```

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



```
- GraphicsArray -
```

Note that 10 is a sink and hence no edges are traversed in a depth−firsttraversal starting from 10.

```
DepthFirstTraversal[s = SetGraphOptions[t, EdgeDirection -> On], 10, Edge]
```

```
{}
```

NOTES

* DepthFirstTraversal works fine for directed graphs as well.

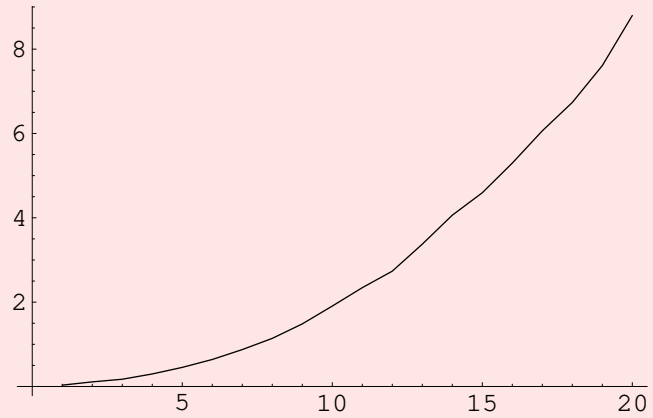
TIMING DSCUSSION

DFS can be easily implmented so that it runs in O(m+n) time on an n−vertex,m−edgegraph. Here are some timing experiments to determine if this is true. The plot below shows that we probably have a quadratic implementation of DFS. The most likely reason for this state of affairs is the two AppendTo[...] calls in DFS. Each is appending to a list that grows linearly and since AppendTo[...] takes linear time in *Mathematica*, we end up with a function that takes quadratic time. This is unfortunate because DFS is fundamental to the package.

```
gt = Table[ GridGraph[20, 10 i], {i, 20}];
rt = Table[ Timing[DepthFirstTraversal[gt[[i]], 1];], {i, 20}]
```

```
{{0.032 Second, Null}, {0.109 Second, Null},
 {0.172 Second, Null}, {0.297 Second, Null}, {0.453 Second, Null},
 {0.64 Second, Null}, {0.875 Second, Null}, {1.141 Second, Null},
 {1.484 Second, Null}, {1.907 Second, Null}, {2.343 Second, Null},
 {2.735 Second, Null}, {3.375 Second, Null}, {4.062 Second, Null},
 {4.594 Second, Null}, {5.297 Second, Null}, {6.062 Second, Null},
 {6.735 Second, Null}, {7.609 Second, Null}, {8.797 Second, Null}}
```

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




```
- Graphics -
```

To make sure that my conjecture about AppendTo[...] being the problem is true, I reproduce the DFS functions below with the AppendTo[...] deleted. As a result, the new DFS function does a depth first traversal of the graph, but does not remember the edges or the vertices visited in order. The evidence is conclusive.

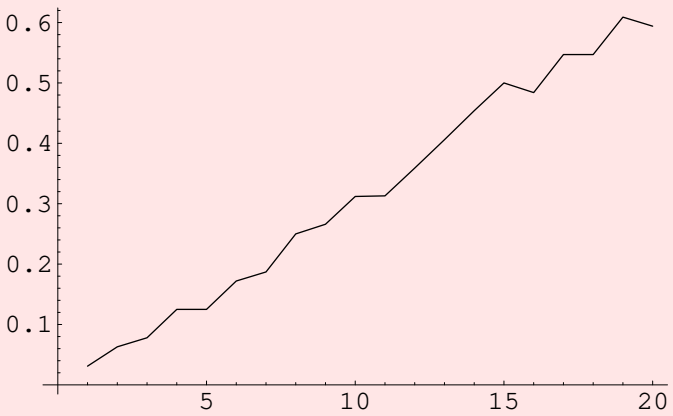
```
NewDepthFirstTraversal[g_Graph, start_Integer, flag_: Vertex] :=
 Block[{visit = {}, e = ToAdjacencyLists[g],
   edges = {}, dfi = Table[0, {V[g]}], cnt = 1}, NewDFS[start];
  If[flag === Edge, edges, visit]]
```

```
NewDFS[v_Integer] := (dfi[[v]] = cnt++;
  Scan[(If[dfi[[#]] == 0, NewDFS[#]]) &, e[[v]]])
```

```
gt = Table[ GridGraph[20, 10 i], {i, 20}];
rt = Table[ Timing[NewDepthFirstTraversal[gt[[i]], 1];], {i, 20}]
```

```
{{0.031 Second, Null}, {0.063 Second, Null},
 {0.078 Second, Null}, {0.125 Second, Null}, {0.125 Second, Null},
 {0.172 Second, Null}, {0.187 Second, Null}, {0.25 Second, Null},
 {0.266 Second, Null}, {0.312 Second, Null}, {0.313 Second, Null},
 {0.359 Second, Null}, {0.406 Second, Null}, {0.454 Second, Null},
 {0.5 Second, Null}, {0.484 Second, Null}, {0.547 Second, Null},
 {0.547 Second, Null}, {0.609 Second, Null}, {0.594 Second, Null}}
```

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




```
- Graphics -
```

How does the new implementation compare in running time with the old version? Here are some more timing experiments to determine this.

```
gt = Table[ DiscreteMath`OldCombinatorica`GridGraph[20, 10 i], {i, 5}];
rt = Table[ Timing[
   DiscreteMath`OldCombinatorica`DepthFirstTraversal[gt[[i]], 1];], {i, 5}]
```

```
{{0.203 Second, Null}, {0.781 Second, Null},
 {1.766 Second, Null}, {3.125 Second, Null}, {4.906 Second, Null}}
```

The first 5 timings for the new implementation of DFS are: {0.031 Second,Null},{0.047 Second,Null},{0.094 Second,-Null},{0.125

Second,Null},{0.125 Second,Null}. So the gains are significant.

TO DO

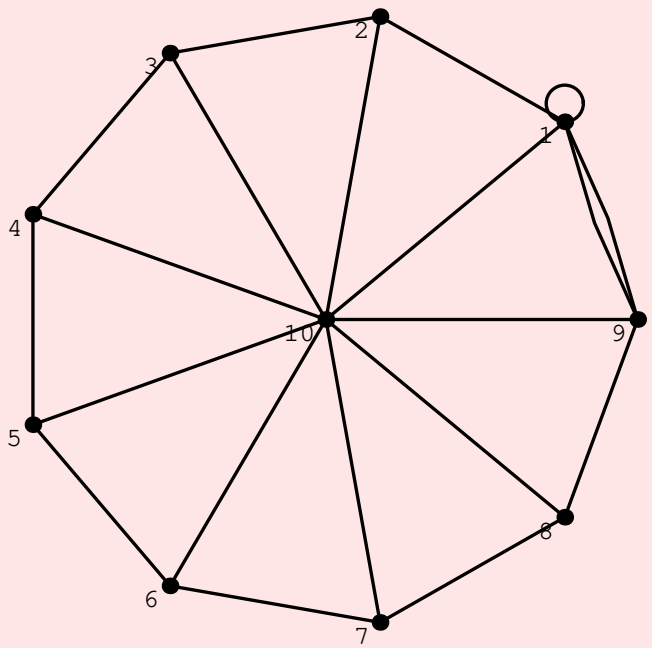
* Modify DFS so that its running time is indeed linear –asymptotically.

BreadthFirstTraversal

```
? BreadthFirstTraversal
```

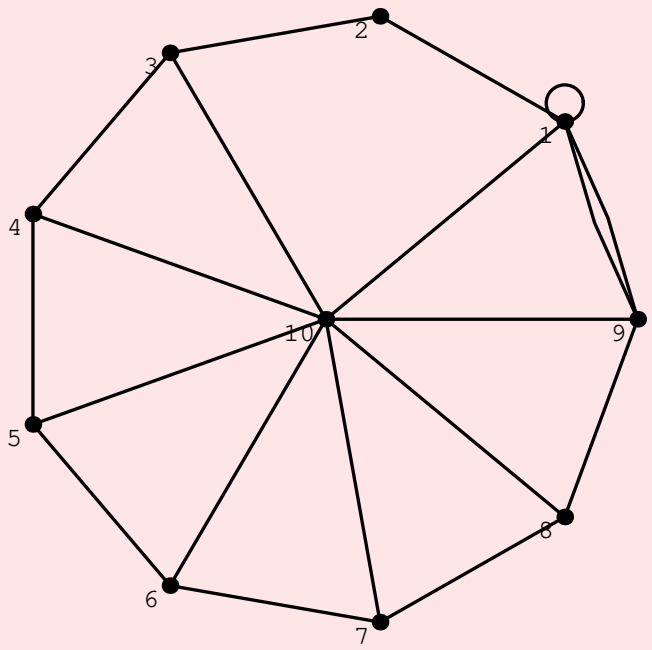
```
BreadthFirstTraversal[g, v] performs a breadth-first traversal
  of graph g starting from vertex v, and gives the breadth-
  first numbers of the vertices. BreadthFirstTraversal[g, v,
  Edge] returns the edges of the graph that are traversed by
  breadth-first traversal. BreadthFirstTraversal[g,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

```
ShowGraph[s = DeleteEdges[t, {{2, 3}}], VertexNumber -> On]
```



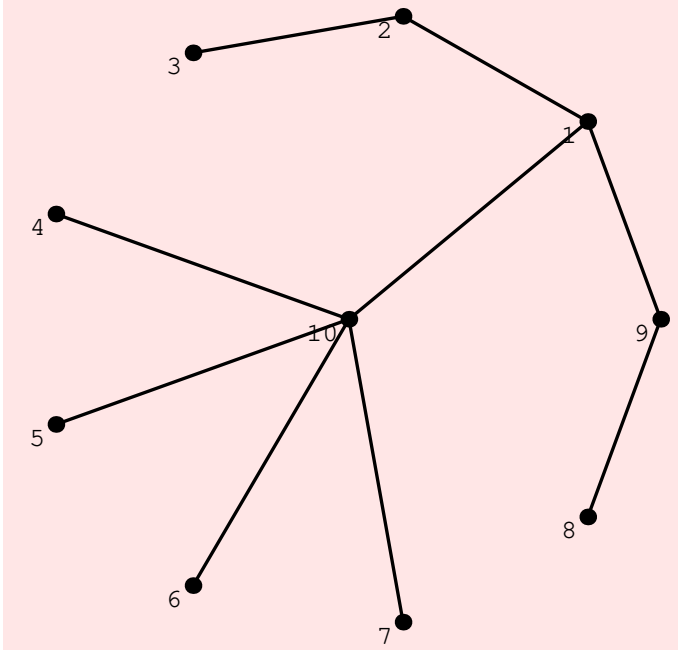

```
- Graphics -
```

```
ShowGraph[s = DeleteEdges[s, {{2, 10}}], VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[BreadthFirstTraversal[s, 1, Tree], VertexNumber -> On]
```




```
- Graphics -
```

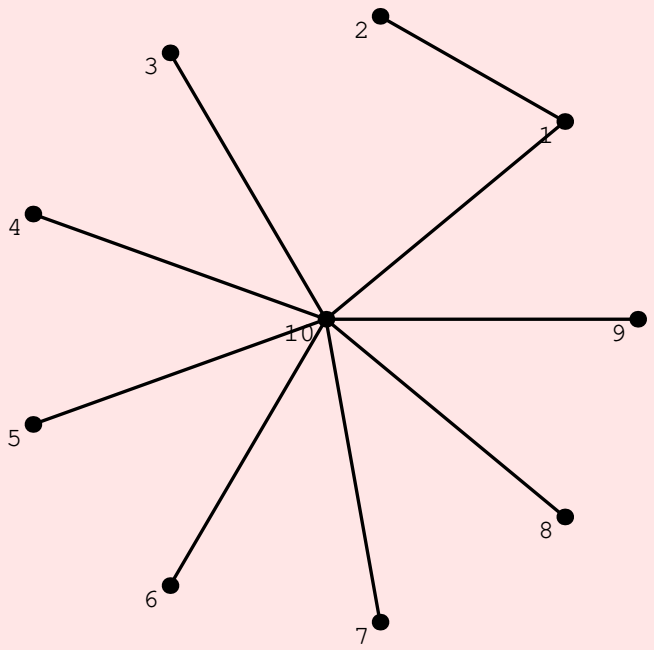
```
BreadthFirstTraversal[s, 1, Edge]
```

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

```
BreadthFirstTraversal[s, 1, Level]
```

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

```
ShowGraph[BreadthFirstTraversal[s, 10, Tree], VertexNumber -> On]
```




```
- Graphics -
```

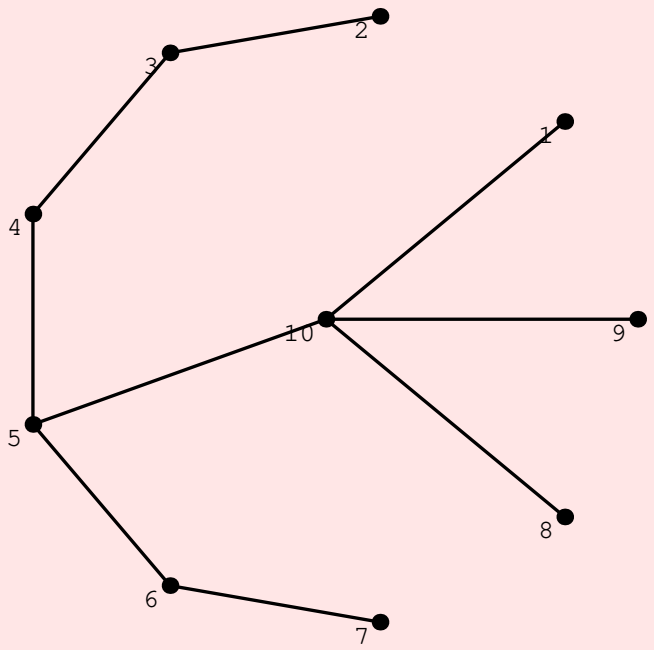
```
BreadthFirstTraversal[s, 10, Edge]
```

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

```
BreadthFirstTraversal[s, 10, Level]
```

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

```
ShowGraph[BreadthFirstTraversal[s, 5, Tree], VertexNumber -> On]
```




```
- Graphics -
```

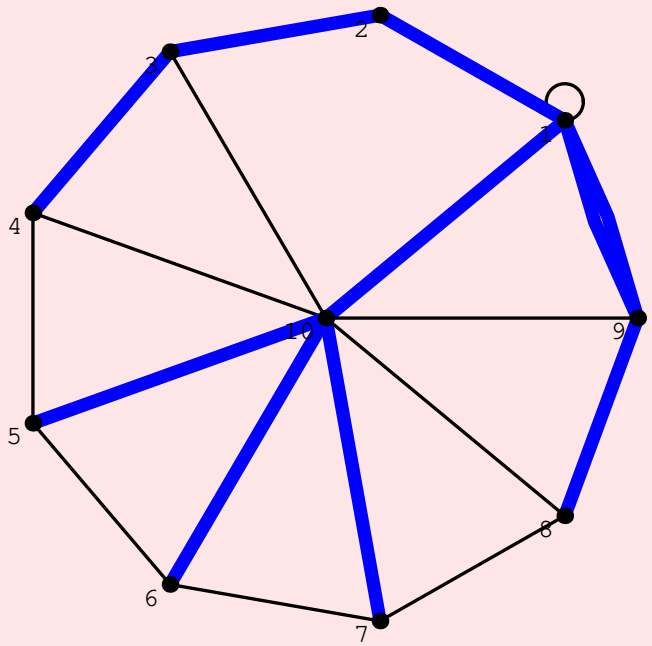
```
BreadthFirstTraversal[s, 5, Edge]
```

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

```
BreadthFirstTraversal[s, 5, Level]
```

```
{2, 3, 2, 1, 0, 1, 2, 2, 2, 1}
```

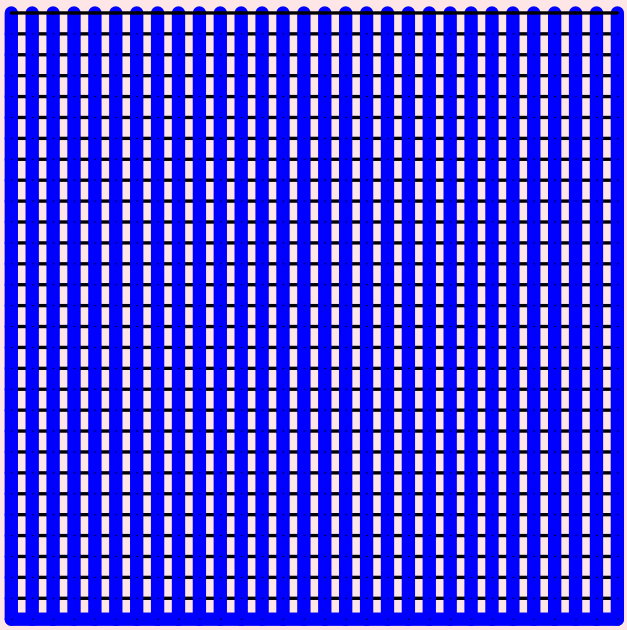
```
ShowGraph[Highlight[s, {Map[Sort, BreadthFirstTraversal[s, 2, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexNumber -> On]
```




```
- Graphics -
```

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

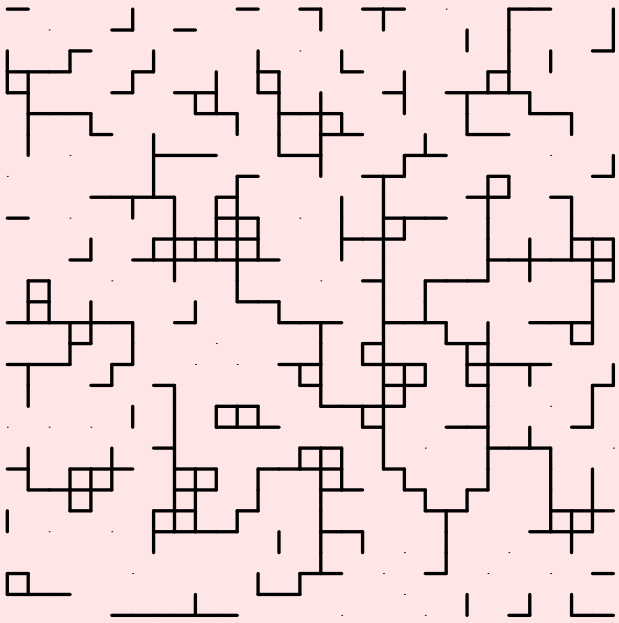
```
ShowGraph[Highlight[s, {Map[Sort, BreadthFirstTraversal[s, 2, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexStyle -> Disc[0]]
```



```
- Graphics -
```

```
s = InduceSubgraph[s, RandomSubset[Range[900]]];
```

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

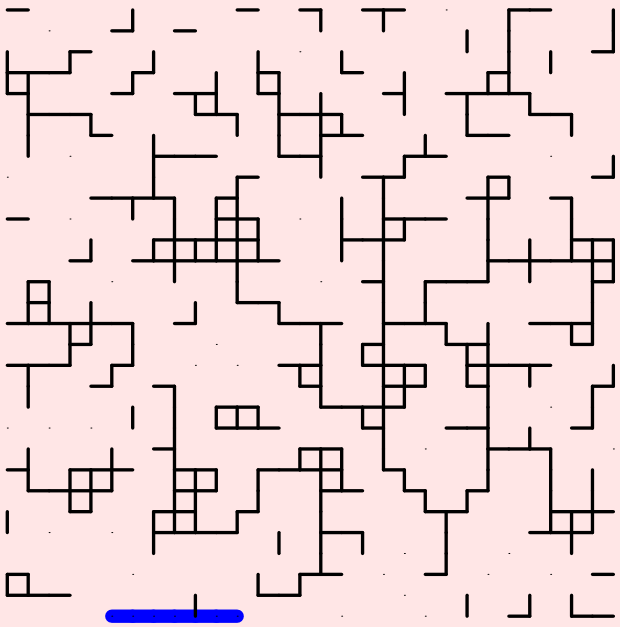


```
- Graphics -
```

```
BreadthFirstTraversal[s, 2, Edge]
```

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

```
ShowGraph[Highlight[s, {Map[Sort, BreadthFirstTraversal[s, 2, Edge]]},
  HighlightedEdgeColors -> {Blue}], VertexStyle -> Disc[0]]
```



```
- Graphics -
```

NOTES

* The eccentricity of a vertex in an unweighted graph can now be easily calculated using the levels returned by BFS. I have rewritten the function Eccentricity[...] in the package to first test if a graph is weighted and then use BFS or Shortest-Paths accordingly.

* It is also easy to partion the edges of the graph into same–leveledges, tree edges, and other edges. This can be used for testing bipartiteness and also two coloring a graph if possible.

TO DO

Two other things that can be done to make BFS and related functions more "fun" and accessible are:

(1) Add an option to BreadthFirstSearch that returns the edge partition mentioned above.

(2) Add an option to BFS that returns an embedding that highlights the induced levels. This can be done easily using RankedEmbedding.

ShowGraph

This is illustrated in a separate notebook.

ShowLabeledGraph

Does not exist. This brings up the question of how we'll provide support for obsolete functions? Do we keep them in this version and then throw them away in future versions?

CircularVertices

```
?CircularVertices
```

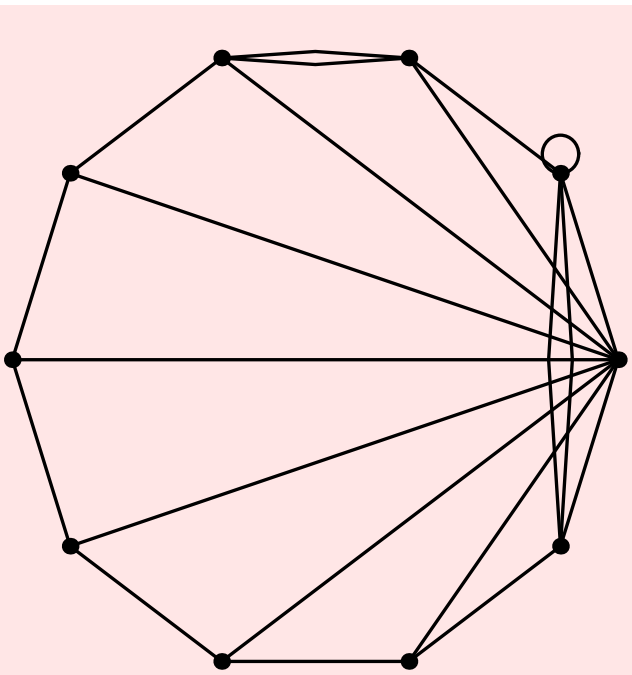
```
CircularVertices[n] constructs a list of n
  points equally spaced on a circle. CircularVertices[
  g] embeds the vertices of g equally spaced on a circle.
```

TO DO

A linear embedding in which the edges are shown as circular arcs above or below the line of vertices should be added to the package.

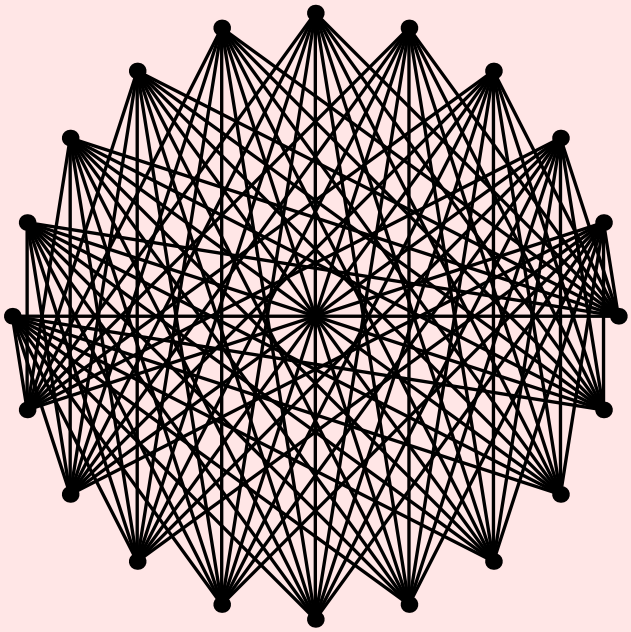
```
s = CircularVertices[t];
```

```
ShowGraph[s]
```



```
- Graphics -
```

```
ShowGraph[CircularVertices[CompleteGraph[10, 10]]]
```



```
- Graphics -
```

RankGraph

```
?RankGraph
```

```
RankGraph[g, l] partitions the vertices into classes based
  on the shortest geodesic distance to a member of list l.
```

TO DO

The code here is essentially the bfs code. It is a bad idea to rewrite this code again. NewRankGraph should be calling bfs. This is yet another reason to make bfs more powerful and more general −with several additional options, possibly.

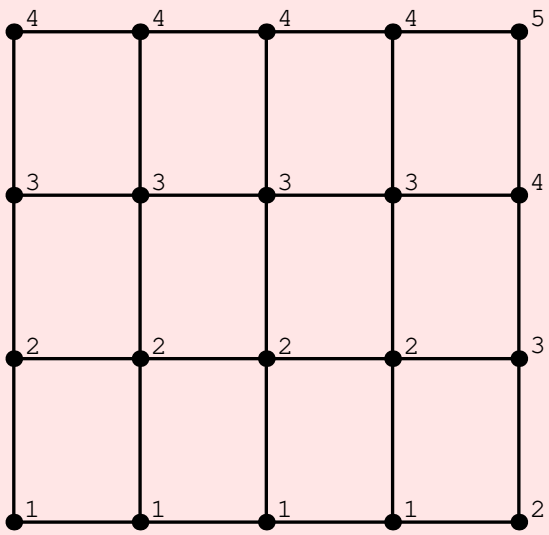
```
s = GridGraph[5, 4];
```

```
r = RankGraph[s, {1, 2, 3, 4}]
```

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

```
s1 = SetGraphOptions[s, Table[{i, VertexLabel -> {r[[i]]}}, {i, Length[r]}]];
```

```
ShowGraph[s1, PlotRange -> Large[0.2]]
```




```
- Graphics -
```

RankedEmbedding

```
?RankedEmbedding
```

```
RankedEmbedding[g, l] performs a ranked embedding of graph g, with the
  vertices ranked in terms of geodesic distance from a member of list l.
```

TO DO

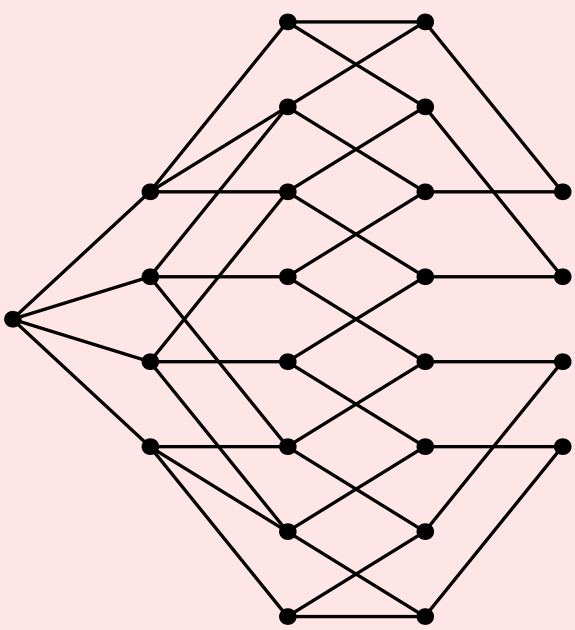
(1) To be consistent with RankedEmbedding, CircularVertices should be called CircularEmbedding.

(2) To be consistent with CircularEmbedding, RankedVertices should have the same name as RankedEmbedding.

(3) RankGraph, in my view, should dissapear and should just be a call to bfs with certain options.

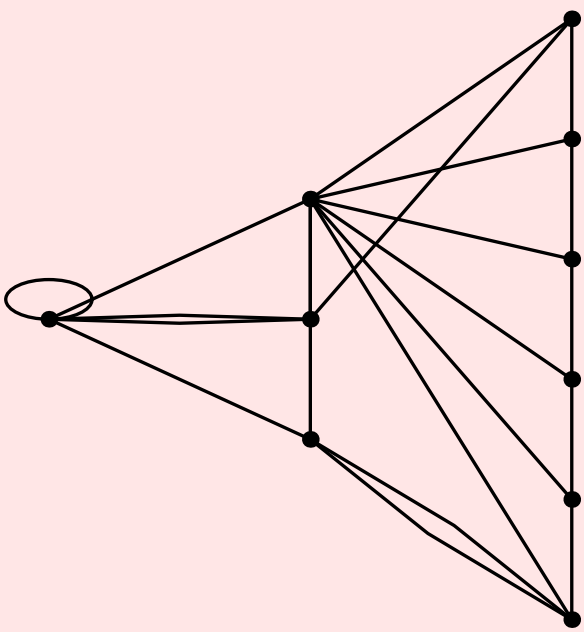
```
s = GridGraph[5, 5];
```

```
ShowGraph[RankedEmbedding[s, {13}]]
```



```
- Graphics -
```

```
ShowGraph[RankedEmbedding[t, {1}]]
```



```
- Graphics -
```

Eccentricity

```
?Eccentricity
```

```
Eccentricity[g] gives the eccentricity of each vertex v
  of graph g, the length of the longest shortest path from v.
```

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

```
Eccentricity[g]
```

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

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

```
Eccentricity[ g ]
```

```
{18, 17, 16, 15, 14, 14, 15, 16, 17, 18, 17, 16, 15, 14, 13, 13, 14, 15, 16, 17, 16,
 15, 14, 13, 12, 12, 13, 14, 15, 16, 15, 14, 13, 12, 11, 11, 12, 13, 14, 15, 14,
 13, 12, 11, 10, 10, 11, 12, 13, 14, 14, 13, 12, 11, 10, 10, 11, 12, 13, 14, 15,
 14, 13, 12, 11, 11, 12, 13, 14, 15, 16, 15, 14, 13, 12, 12, 13, 14, 15, 16, 17,
 16, 15, 14, 13, 13, 14, 15, 16, 17, 18, 17, 16, 15, 14, 14, 15, 16, 17, 18}
```

```
Timing[ Eccentricity[g];]
```

```
{1.984 Second, Null}
```

```
g = SetEdgeWeights[g];
```

```
Timing[ Eccentricity[g]; ]
```

```
{3.844 Second, Null}
```

NOTES

* In general computing eccentricities of vertices in a graph is time consuming –typically cubic (for dense graphs). However, the eccentricites in an unweighted graph are much faster to compute than those in a weighted graph since BFS can be used instead of shortest path algorithms. I have separated the computation of Eccentricites of weighted and unweighted graphs –the difference is significant.

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

```
Timing[ DiscreteMath`OldCombinatorica`Eccentricity[g]; ]
```

```
{6.016 Second, Null}
```

Diameter

```
?Diameter
```

```
Diameter[g] gives the diameter of graph g, the length
  of the longest shortest path between two vertices of g.
```

```
g = CompleteGraph[20];
```

```
Diameter[g]
```

```
1
```

```
g = RandomGraph[25, .3];
```

```
Diameter[g]
```

```
4
```

```
g = RandomGraph[50, .3];
```

```
Diameter[g]
```

```
3
```

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

```
Diameter[g]
```

```
2
```

TO DO

For fixed p, as n–>Infinitythe expected diameter of the random graph goes to 2. I should expriment with this some more.

Radius

```
?Radius
```

```
Radius[g] gives the radius of graph
  g, the minimum eccentricity of any vertex of g.
```

```
g = Wheel[100];
```

```
Radius[g]
```

```
1
```

GraphCenter

```
?GraphCenter
```

```
GraphCenter[g] gives a list of the
  vertices of graph g with minimum eccentricity.
```

```
g = Wheel[100];
```

```
GraphCenter[g]
```

```
{100}
```

RadialEmbedding

```
?RadialEmbedding
```

```
RadialEmbedding[g] constructs a radial embedding
  of graph g, radiating from the center of the graph.
```

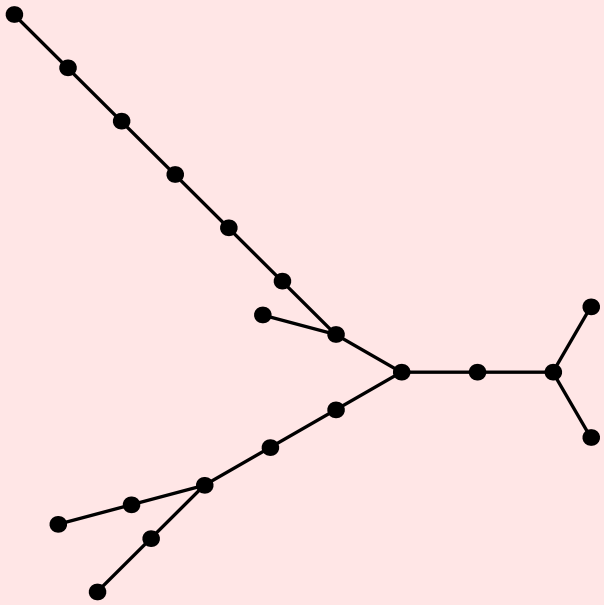
TO DO

(1) To be consistent with CircularEmbedding and RankedEmbedding, RadialEmbedding should also have a version that simply returns the embedding of the vertices.

(2) This remark actually applies to the functions above −Eccentricity, Diameter, Radius, and GraphCenter. For trees, the general form of the functions should not be used. It is easy to calculate the graph center of a tree −can be done in linear time. So that is what should be used.

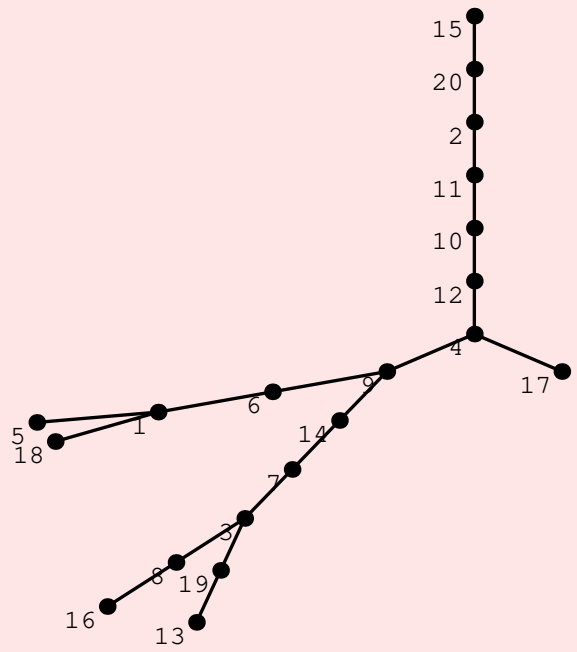
```
s = RandomTree[20];
```

```
ShowGraph[s]
```



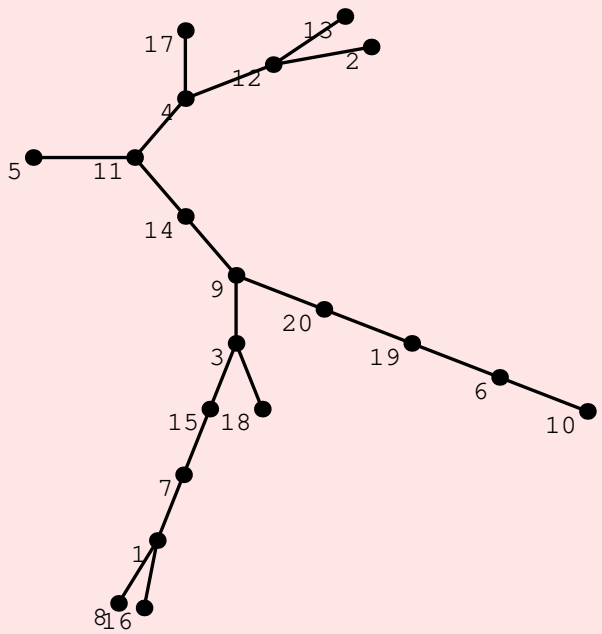
```
- Graphics -
```

```
ShowGraph[RadialEmbedding[s, 11], VertexNumber -> On]
```



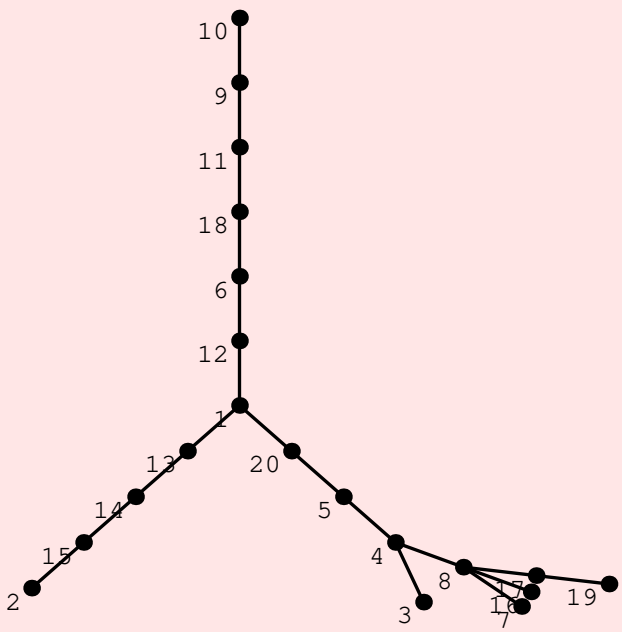

```
- Graphics -
```

```
ShowGraph[RadialEmbedding[RandomTree[20], 11], VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[RadialEmbedding[RandomTree[20], 11], VertexNumber -> On]
```

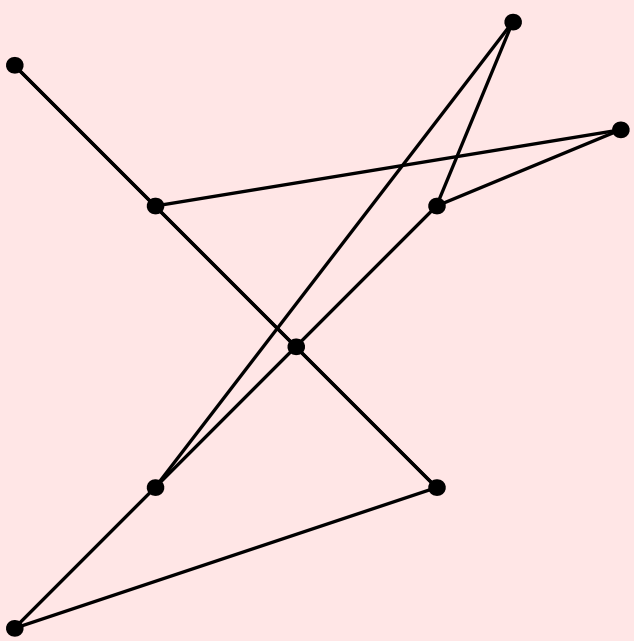



```
- Graphics -
```

NOTES

We can get strange embeddings as shown below by constructing RadialEmbeddings of graphs that are not trees.

```
ShowGraph[RadialEmbedding[ GridGraph[3, 3]]]
```



```
- Graphics -
```

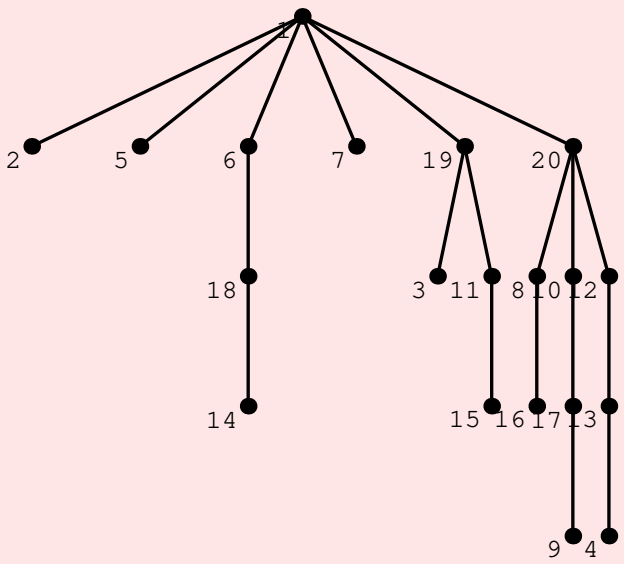
RootedEmbedding

```
?RootedEmbedding
```

```
RootedEmbedding[g, v] constructs a
  rooted embedding of graph g with vertex v as the root.
```

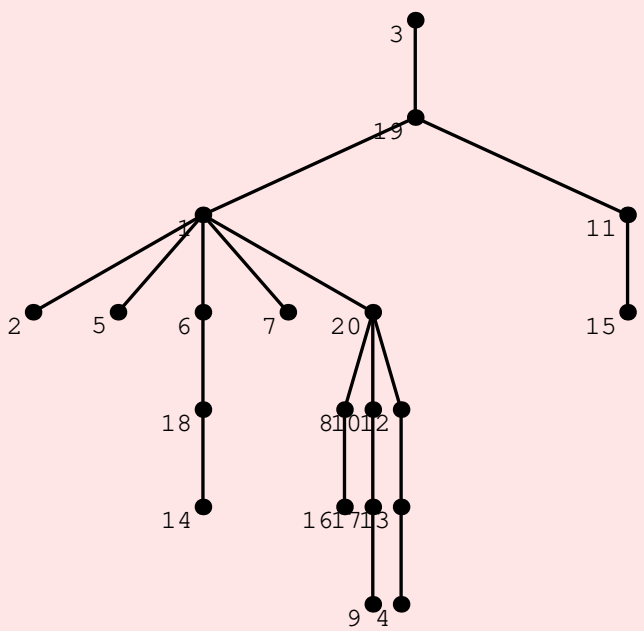
```
s = RandomTree[20];
```

```
ShowGraph[RootedEmbedding[s, 1], VertexNumber -> On, PlotRange -> Large[0.05]]
```



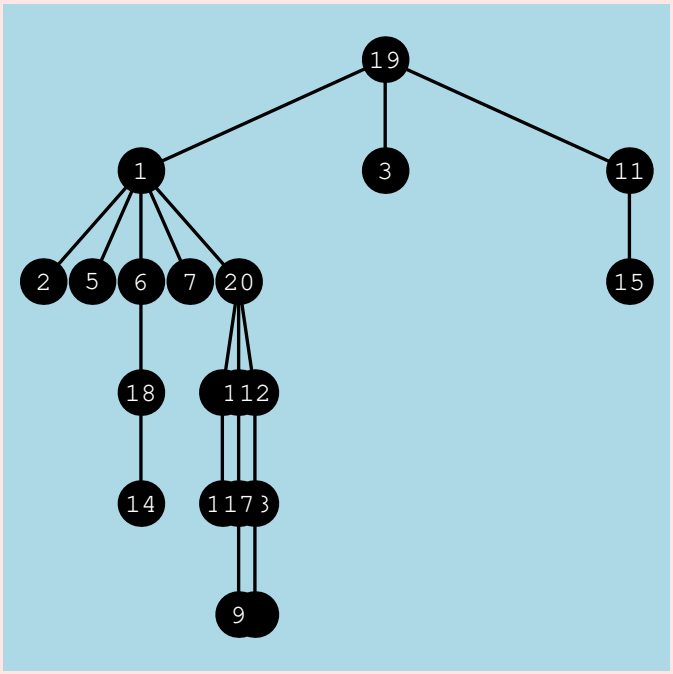

```
- Graphics -
```

```
ShowGraph[RootedEmbedding[s, 3], VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[RootedEmbedding[s, 19], VertexNumber -> Center,
 VertexNumberColor -> White, VertexStyle -> Disc[Large],
 PlotRange -> Large[0.03], Background -> LightBlue]
```




```
- Graphics -
```

TranslateVertices

```
?TranslateVertices
```

```
TranslateVertices[v, {x, y}] adds the vector {x, y} to the vertex
  embedding location of each vertex in list v. TranslateVertices[g,
  {x, y}] translates the embedding of the graph g by the vector {x, y}.
```

TO DO

TranslateVertices will be more useful when subsets of vertices can be translated. I should generalize the function to be able to do this.

```
t
```

```
-Graph:<21, 10, Undirected>-
```

```
TranslateVertices[t, {.001, .002}]
```

```
-Graph:<21, 10, Undirected>-
```

DilateVertices

```
?DilateVertices
```

```
DilateVertices[v, d] multiplies each coordinate of each vertex position
  in list l by d, thus dilating the embedding. DilateVertices[
  g, d] dilates the embedding of the graph g by the factor d.
```

TO DO

The user should be able to dilate subsets of vertices as well.

```
t
```

```
-Graph:<21, 10, Undirected>-
```

```
DilateVertices[t, 2]
```

```
-Graph:<21, 10, Undirected>-
```

RotateVertices

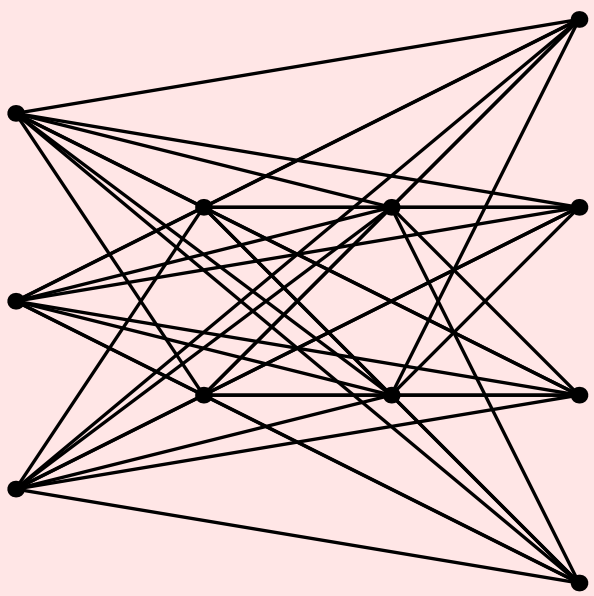
```
?RotateVertices
```

```
RotateVertices[v, theta] rotates each vertex position in list v by theta
  radians around the origin (0, 0). RotateVertices[g, theta] rotates
  the embedding of the graph g by theta radians about the origin (0, 0).
```

TO DO

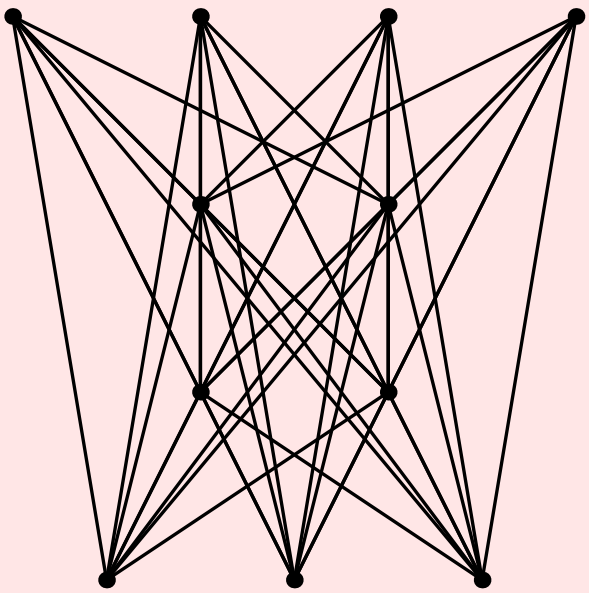
Same as for the previous 2 functions. We should be able the Rotate a subset of the vertices.

```
ShowGraph[s = CompleteGraph[3, 2, 2, 4]]
```



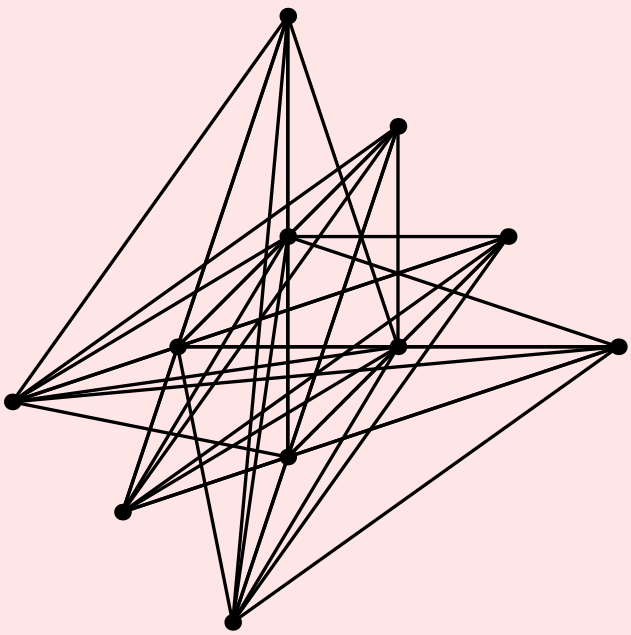
```
- Graphics -
```

```
ShowGraph[RotateVertices[s, Pi / 2]]
```



```
- Graphics -
```

```
ShowGraph[RotateVertices[s, Pi / 4]]
```



```
- Graphics -
```

NormalizeVertices

```
?NormalizeVertices
```

```
NormalizeVertices[v] gives a list of
  vertices with a similar embedding as v but with all
  coordinates of all points scaled to be between 0 and 1.
```

```
s = CompleteGraph[7]
```

```
-Graph:<21, 7, Undirected>-
```

```
Vertices[s]
```

```
{{0.62349, 0.781831}, {-0.222521, 0.974928}, {-0.900969, 0.433884},
 {-0.900969, -0.433884}, {-0.222521, -0.974928}, {0.62349, -0.781831}, {1., 0}}
```

```
s = TranslateVertices[s, {0.5, 0.3}]
```

```
-Graph:<21, 7, Undirected>-
```

```
Vertices[s]
```

```
{{1.12349, 1.08183}, {0.277479, 1.27493},
 {-0.400969, 0.733884}, {-0.400969, -0.133884},
 {0.277479, -0.674928}, {1.12349, -0.481831}, {1.5, 0.3}}
```

```
Vertices[NormalizeVertices[s]]
```

```
{{0.826886, 0.807732}, {0.437903, 0.896515}, {0.125962, 0.647751},
 {0.125962, 0.248764}, {0.437903, 0.}, {0.826886, 0.0887829}, {1., 0.448258}}
```

ShakeGraph

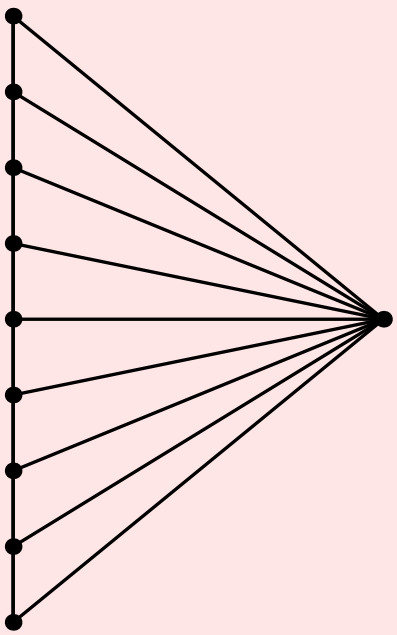
```
? ShakeGraph
```

```
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.
```

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

```
-Graph:<18, 10, Undirected>-
```

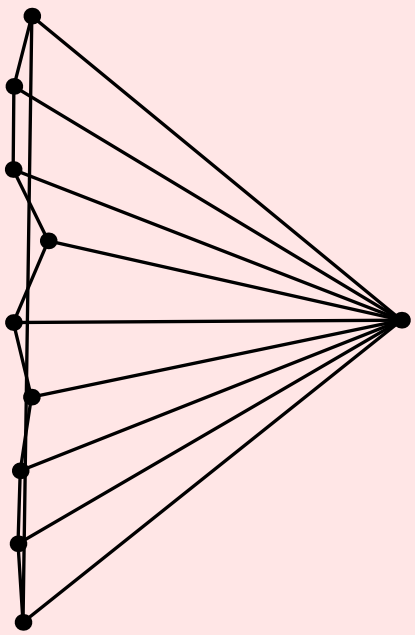
```
ShowGraph[s]
```



```
- Graphics -
```

```
ss = ShakeGraph[s];
```

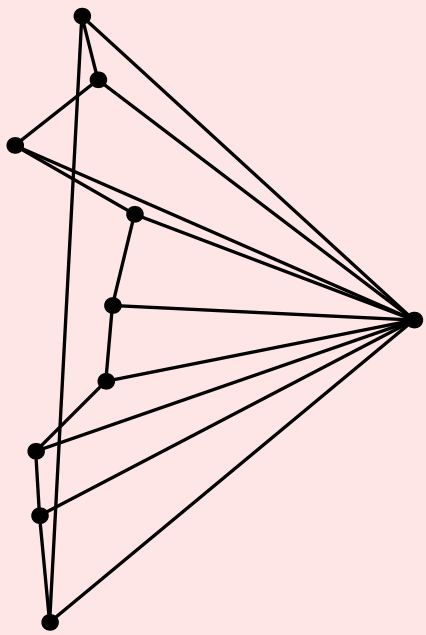
```
ShowGraph[ss]
```



```
- Graphics -
```

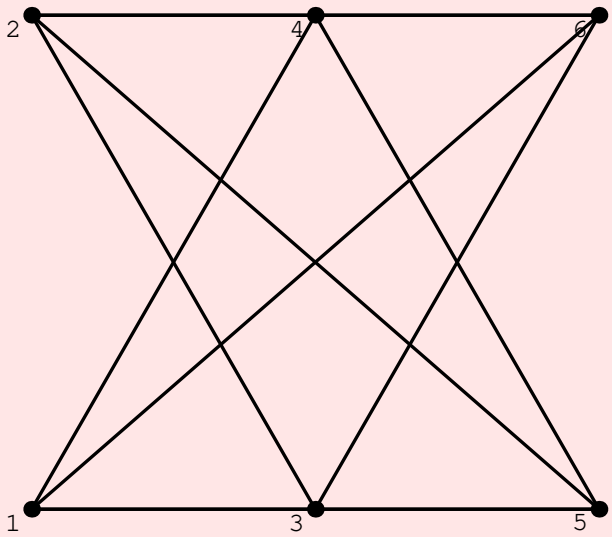
```
ss = ShakeGraph[s, 0.3];
```

```
ShowGraph[ss]
```



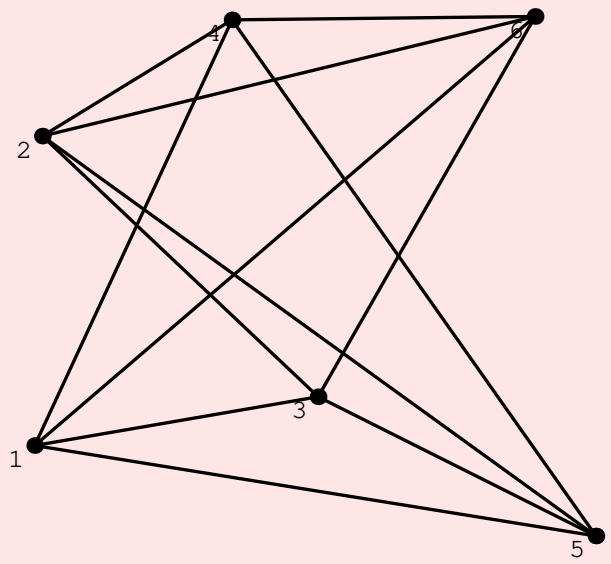
```
- Graphics -
```

```
ShowGraph[CompleteGraph[2, 2, 2], VertexNumber -> On]
```




```
- Graphics -
```

```
ShowGraph[ShakeGraph[CompleteGraph[2, 2, 2], 0.3], VertexNumber -> On]
```




```
- Graphics -
```

GraphOptions

```
? GraphOptions
```

```
GraphOptions[g] returns the display options associated
  with g. GraphOptions[g, v] returns the display options
  associated with vertex v in g. GraphOptions[g, {u, v}]
  returns the display options associated with edge {u, v} in g
```

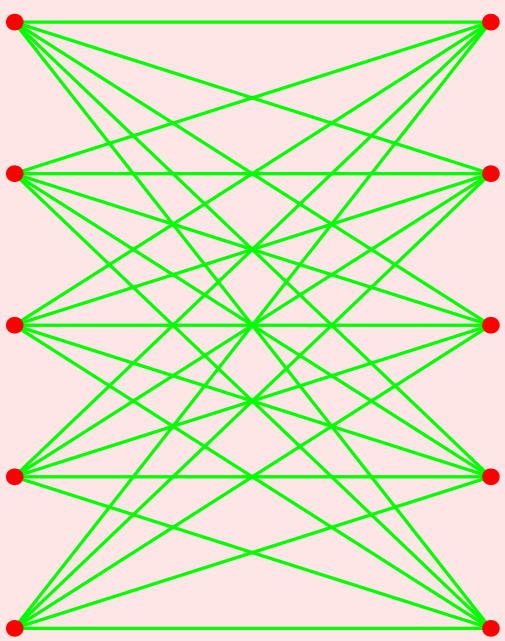
```
GraphOptions[s]
```

```
{}
```

```
r = SetGraphOptions[CompleteGraph[5, 5], VertexColor → Red, EdgeColor → Green]
```

```
-Graph:<25, 10, Undirected>-
```

```
ShowGraph[r]
```



```
- Graphics -
```

```
GraphOptions[r]
```

```
{VertexColor → RGBColor[1., 0., 0.], EdgeColor → RGBColor[0., 1., 0.]}
```

NOTES

* This function's behavior might change when I make changes to ShowGraph.

SpringEmbedding

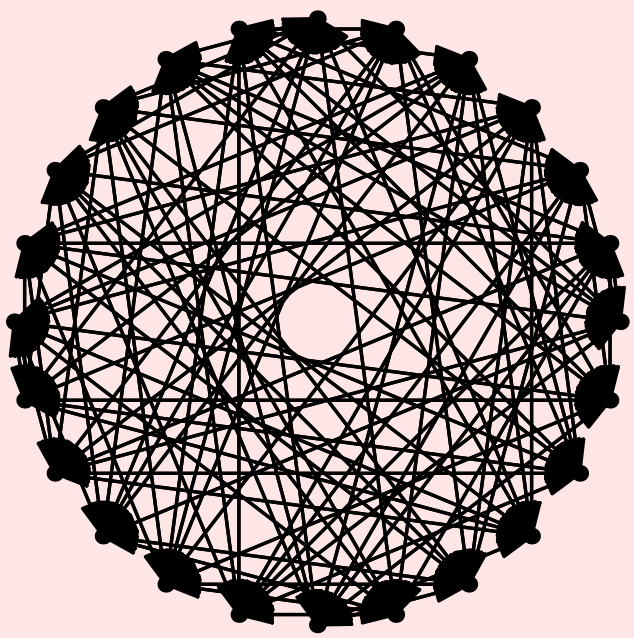
```
? SpringEmbedding
```

```
SpringEmbedding[g] beautifies the embedding of
  graph g by modeling the embedding as a system of springs.
```

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

```
oldS = DiscreteMath`OldCombinatorica`MakeGraph[
   Permutations[{1, 2, 3, 4}],  (Count[#1 - #2, 0] == 0) &];
```

```
ShowGraph[s]
```



```
- Graphics -
```

```
V[s]
```

```
24
```

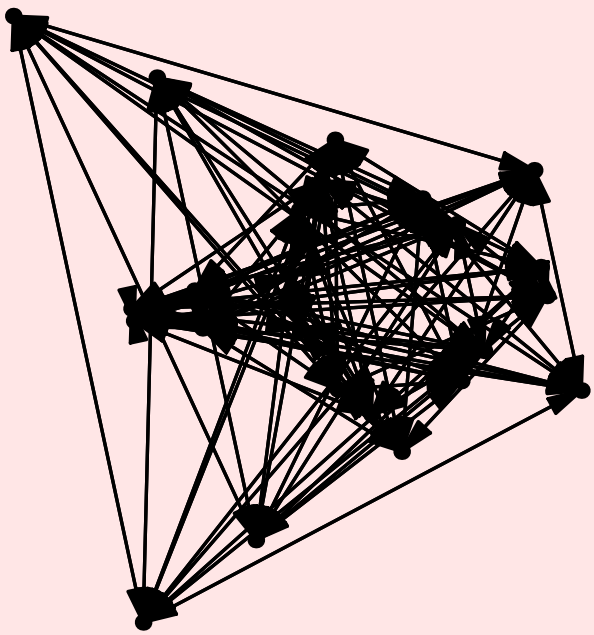
```
M[s]
```

```
216
```

```
Timing[r = SpringEmbedding[s];]
```

```
{2.172 Second, Null}
```

```
ShowGraph[r]
```



```
- Graphics -
```

```
Timing[r = DiscreteMath`OldCombinatorica`SpringEmbedding[oldS];]
```

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

ReadGraph

Not yet implemented. Probably will not be implemented.

WriteGraph

Not yet implemented. Probably will not be implemented.