Introduction to Graph Theory

Graphs and Digraphs

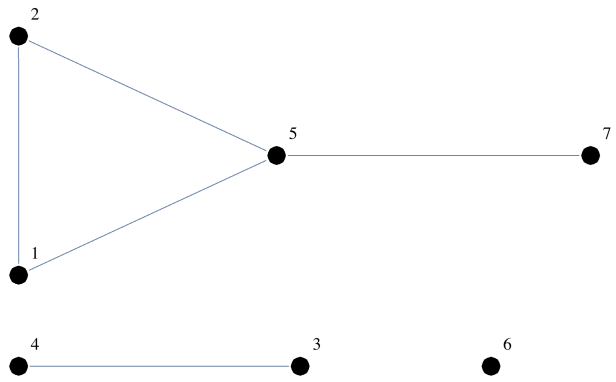
Definition 1.1. If S is a set, let $S^{(n)}$ denote the set of unordered n-tuples (with possible repetition).

Definition 1.2. A ***graph*** $G = (V, E)$ is an ordered pair of finite sets. Elements of V are called ***vertices***, and elements of $E \subseteq V^{(2)}$ are called ***edges***. We refer to V as the ***vertex set*** of G, with E being the ***edge set***. The cardinality of V is called the **order** of G, wriiten as $v(G)$, and the cardinality of E is called the **size** of G, written as $e(G)$.

To simplify the notation, we shall speak of a vertex $u \in G$ instead of $u \in V$, and, similarly, an edge $e \in G$ instead of $e \in E$. If there is an edge between vertices u and v, *Mathematica* uses $u \longmapsto v$ (esc ue esc) to denote the edge between u and v.

Example 1.1. The graph $G = (V, E)$, with $V = \{1, 2, \ldots, 7\}$ and $E = \{1 \longmapsto 2, 1 \longmapsto 5, 2 \longmapsto 5, 3 \longmapsto 4, 5 \longmapsto 7\}$, is shown as follows.

```
g = Graph[Range[7], {1 ⟼ 2, 1 ⟼ 5, 2 ⟼ 5, 3 ⟼ 4, 5 ⟼ 7},
  VertexLabels → "Name", VertexSize → Medium, VertexStyle → Black, ImagePadding → 20]
```




```
{VertexCount[g], EdgeCount[g]}
```

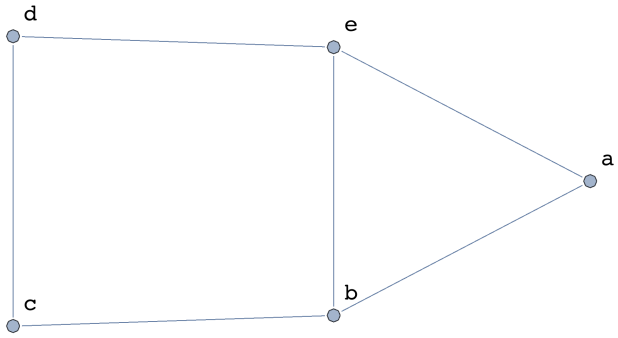
```
{7, 5}
```

A vertex $v \in G$ is **incident** with an edge $e \in G$ if $e = u \longmapsto v$ for some vertex $u \in G$. The two vertices incident with an edge are its endvertices or ends, and an edge joins its ends. Two vertices $u, v \in G$ are **adjacent**, if $u \longmapsto v$ is an edge of G. Two edges $e \neq f$ are adjacent if they have an end in common.

Example 1.2. Consider the graph as shown below.

1. List the vertex and edge sets of the graph.
2. For each vertex, list all vertices that are adjacent to it.
3. Which vertex or vertices have the largest number of adjacent vertices? Similarly, which vertex or vertices have the smallest number of adjacent vertices?

```
g = Graph[{"a" ⟷ "b", "a" ⟷ "e", "b" ⟷ "c", "b" ⟷ "e", "c" ⟷ "d", "d" ⟷ "e"},
  VertexLabels → "Name",
  VertexLabelStyle → Directive[FontFamily → "Courier", 12], ImagePadding → 20]
```



Solution:

```
{VertexList[g], EdgeList[g]}
```

```
{{a, b, e, c, d}, {a ⟷ b, a ⟷ e, b ⟷ c, b ⟷ e, c ⟷ d, d ⟷ e}}
```

```
Table[{v, AdjacencyList[g, v]}, {v, VertexList[g]}]
```

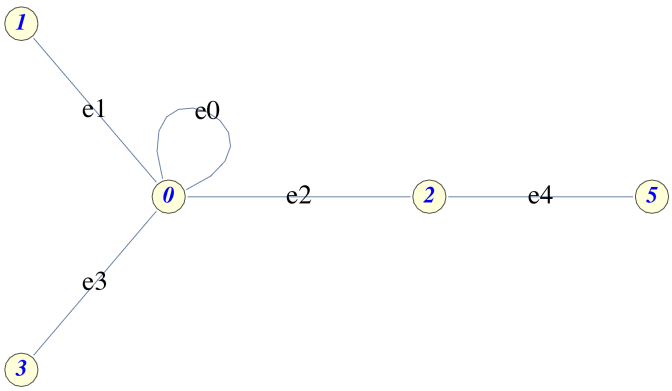
```
{{a, {b, e}}, {b, {a, e, c}}, {e, {a, b, d}}, {c, {b, d}}, {d, {e, c}}}
```

```
Sort[%, Length[#1⟦2⟧] > Length[#2⟦2⟧] &]
```

```
{{e, {a, b, d}}, {b, {a, e, c}}, {d, {e, c}}, {c, {b, d}}, {a, {b, e}}}
```

The definition of a graph allows the possibility of the edge e having identical end vertices. That is, it is possible to have a vertex u joined to itself by an edge—such an edge is called a ***loop***.

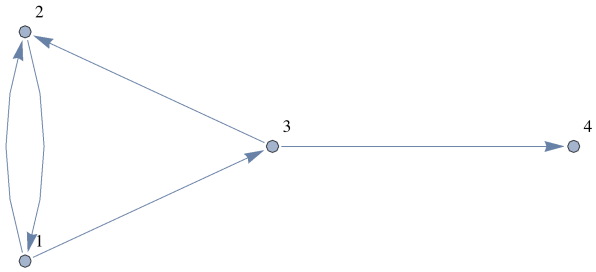
```
g = Graph[{Labeled[0 ⟷ 0, "e0"], Labeled[0 ⟷ 1, "e1"],
    Labeled[0 ⟷ 2, "e2"], Labeled[0 ⟷ 3, "e3"], Labeled[2 ⟷ 5, "e4"]},
 VertexStyle → LightYellow, VertexSize → 0.15,
  VertexLabels → Placed["Name", {1 / 2, 1 / 2}],
  VertexLabelStyle → Directive[12, Blue, Bold, Italic],
  EdgeLabelStyle → Directive[14, Black]]
```



Definition 1.3. A ***digraph*** $G = (V, E)$ is a graph, where the edges have a direction, that is, the edges are ordered $E \subseteq V \times V$.

Mathematica uses $u \longmapsto v$ (esc de esc) to denote the directed edge from u to v. In the directed edge $u \longmapsto v$, u is called the tail vertex, and v is called the head vertex.

```
g = Graph[{1 ↔ 2, 1 ↔ 3, 2 ↔ 1, 3 ↔ 2, 3 ↔ 4},
  VertexLabels → "Name", ImagePadding → 20]
```




```
(* An alternative way to denote the directed edge u↔v in Mathematica is u→v.  *)
Graph[{1 → 2, 1 → 3, 2 → 1, 3 → 2, 3 → 4}]
```

Definition 1.4. Then **indegree**, id(v), of a vertex v in a digraph G is the number of edges such that v is the head of those edges. The **outdegree**, od(v), is the number of edges such that v is the tail of those edges. The **degree**, d(v), of v is the sum of the indegree and the outdegree of v, counting each loop twice.

```
{VertexInDegree[g], VertexOutDegree[g], VertexDegree[g]}
```

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

Theorem 1.1 (Euler 1736). *For any graph G with e edges and n vertices, v_1, ..., v_n, we have*

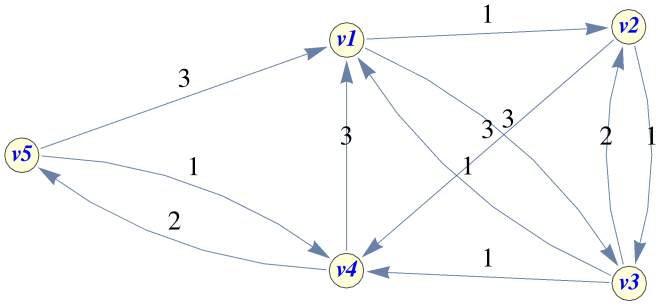
$$\sum_{i=1}^{n} d(v_i) = 2\,e$$

```
{Plus @@ VertexDegree[g], 2 EdgeCount[g]}
```

```
{10, 10}
```

Definition 1.5. Given a graph $G = (V, E)$, a ***weight*** is a function ω that maps the edges of G to the nonnegative real numbers. The graph G together with a weighted function is called a ***weighted graph***.

```
Graph[{"v1" ↔ "v2", "v1" ↔ "v3", "v2" ↔ "v3", "v2" ↔ "v4", "v3" ↔ "v2",
  "v3" ↔ "v1", "v3" ↔ "v4", "v4" ↔ "v1", "v4" ↔ "v5", "v5" ↔ "v1", "v5" ↔ "v4"},
 EdgeWeight → {1, 3, 1, 3, 2, 1, 1, 3, 2, 3, 1},
 VertexStyle → LightYellow, VertexSize → 0.15,
 VertexLabels → Placed["Name", {1 / 2, 1 / 2}],
 VertexLabelStyle → Directive[12, Blue, Bold, Italic],
 EdgeLabels → "EdgeWeight", EdgeLabelStyle → 14]
```



Mathematica also supports vertex weight. Similar to the command `EdgeWeight`, vertex weights can be assigned by the command `VertexWeight`.

Note that *Mathematica* does not support multigraphs. The following graph has two edges from 0 to 2 that cause problem!

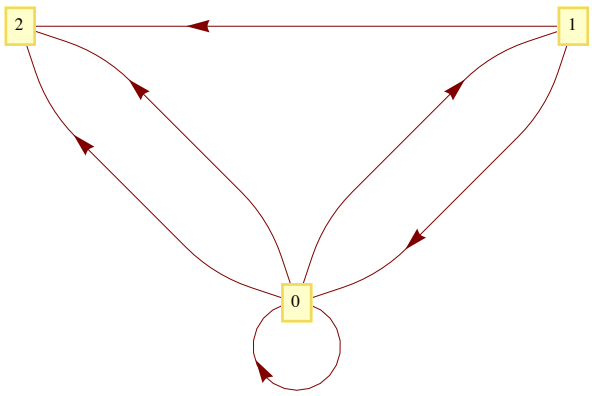
```
h = Graph[{Labeled[0 ↔ 0, "e0"],
    Labeled[0 ↔ 1, "e1"], Labeled[0 ↔ 2, "e2"], Labeled[0 ↔ 2, "e3"],
    Labeled[1 ↔ 2, "e4"], Labeled[1 ↔ 0, "e5"]}, DirectedEdges → True]
```

Graph::supp : Mixed graphs and multigraphs are not supported. ≫

$$\text{Graph}\left[\left\{\underset{e0}{0 \leftrightarrow 0},\ \underset{e1}{0 \leftrightarrow 1},\ \underset{e2}{0 \leftrightarrow 2},\ \underset{e3}{0 \leftrightarrow 2},\ \underset{e4}{1 \leftrightarrow 2},\ \underset{e5}{1 \leftrightarrow 0}\right\}, \text{DirectedEdges} \to \text{True}\right]$$

However, we can still draw the graph, if we use the command `GraphPlot` instead. Not like the command `Graph` returns a `Graph` object (`Graph` objects are atoms in *Mathematica*), `GraphPlot` returns a `Graphics` object.

```
GraphPlot[{0 → 0, 0 → 1, 0 → 2, 0 → 2, 1 → 2, 1 → 0}, DirectedEdges → True,
 SelfLoopStyle → 0.2, VertexCoordinateRules → {0 → {0, 0}, 2 → {-1, 1}, 1 → {1, 1}},
 MultiedgeStyle → 0.2, VertexLabeling → True]
```

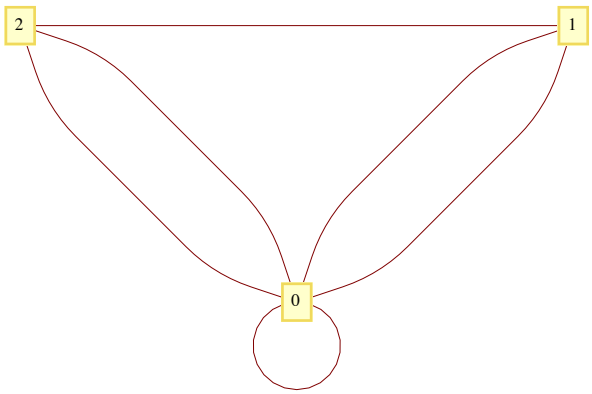



```
Head[%]
```

```
Graphics
```

The default value of `DirectedEdges` is False.

```
GraphPlot[{0 → 0, 0 → 1, 0 → 2, 0 → 2, 1 → 2, 1 → 0}, SelfLoopStyle → 0.2,
 VertexCoordinateRules → {0 → {0, 0}, 2 → {-1, 1}, 1 → {1, 1}},
 MultiedgeStyle → 0.2, VertexLabeling → True]
```



Simple Graphs and Degree Sequences

Definition 2.1. A **simple** graph is a graph with no self-loops and no multiple edges.

Notational Convention: Unless stated otherwise, all graphs are simple graphs in this note.

Definition 2.2. Let G be a graph with n vertices. The **degree sequence** of G is the ordered n-tuple of the vertex degrees of G arranged in the nonincreasing order.

In[20]:=
```
(* Mathematica does not have a command to prodce degree sequence *)
degreeSequence[g_] := Reverse[Sort[VertexDegree[g]]]
```

```
g = Graph[Range[7], {1 ↔ 2, 1 ↔ 5, 2 ↔ 5, 3 ↔ 4, 5 ↔ 7}]
degreeSequence[g]
```

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

```
CompleteGraph[10]
degreeSequence[CompleteGraph[10]]
```

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

Let S be a nonincreasing sequence of nonnegative integers. Then S is said to be *graphical* if it is the degree sequence of some graph. If G is a graph with degree sequence S, we say G *realizes* S.

Theorem 2.1 (Havel 1955 & Hakimi 1962) *Consider the nonincreasing sequence* $S_1 = (d_1, d_2, \ldots, d_n)$ *of nonnegative integers, where* $n \geq 2$ *and* $d_1 \geq 1$. *Then* S_1 *is graphical if and only if the sequence*

$$S_2 = (d_2 - 1,\ d_3 - 1,\ \ldots,\ d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$$

is graphical.

Havel Hakimi Algorithm
Input: A nonincreasing sequence $S = \{d_1, d_2, \ldots, d_n\}$ of integers, where $n \geq 1$.
Output: True if S is graphical; False otherwise.
1: if $\text{Max}(S) \geq \text{length}(S)$, then returns False
2: if $\sum d_i$ is odd then returns False
3: if $\min(S) < 0$, then retruns False
4: if $\max(S) = 0$, then returns True
5: $S \leftarrow \left(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_{\text{length}(S)}\right)$
6: sort S in nonincreasing order, and go to step 3.

In[21]:=
```
(* An implementation of Havel Hakimi Algorithm *)
graphicalQ[s_List] := (First[s] == 0) /; (Length[s] == 1)
graphicalQ[s_List] := False /; (Min[s] < 0) || (Max[s] ≥ Length[s])
graphicalQ[s_List] := False /; OddQ[Plus @@ s]
graphicalQ[s_List] :=
 Module[{m, sorted = Reverse[Sort[s]]},
   m = First[sorted];
   graphicalQ[Join[Take[sorted, {2, m + 1}] - 1, Drop[sorted, m + 1]]]
 ]
```

The following code generate a graph satisfying the given degree squence.

```
realizeDegreeSequence[s_List] :=
 If[graphicalQ[s], RandomGraph[DegreeGraphDistribution[s]]]
```

In[25]:=
```
graphicalQ[degreeSequence[RandomGraph[{10, 5}]]]
```

Out[25]=
```
True
```

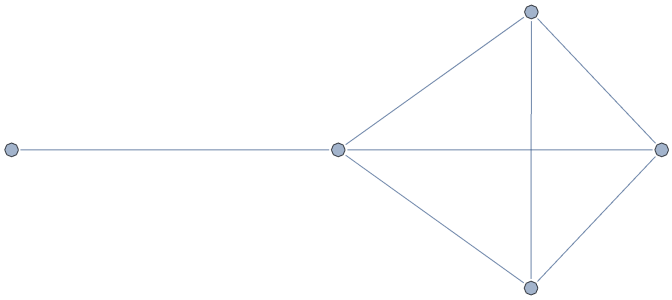
```
graphicalQ[{4, 3, 3, 3, 1}]
```

```
True
```

```
graphicalQ[{7, 6, 5, 4, 3, 2, 1}]
```

```
False
```

```
realizeDegreeSequence[{4, 3, 3, 3, 1}]
```



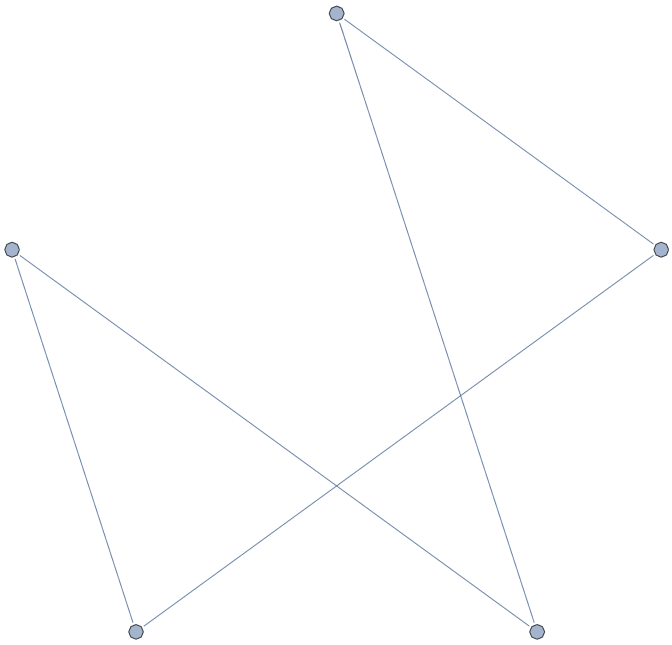
```
(* Gives the adjacency list of all vertices of the given graph   *)
adjacencyLists[g_] := AdjacencyList[g, #] & /@ VertexList[g]
```

```
(* gives the edges of the undirected graph from its adjacency list *)
fromAdjacencyLists[lst_] := Module[
  {vertexNum = Length[lst], vertexAdjacencyLists, totalEdgePairs, edgePairs},
  vertexAdjacencyLists = {Range[vertexNum], lst} // Transpose;
  totalEdgePairs = Flatten[Outer[List, {#[[1]]}, #[[2]]] & /@ vertexAdjacencyLists, 2];
  edgePairs = Union[Sort[#] & /@ totalEdgePairs];
  edgePairs /. List[x_, y_] → UndirectedEdge[x, y]
 ]
```

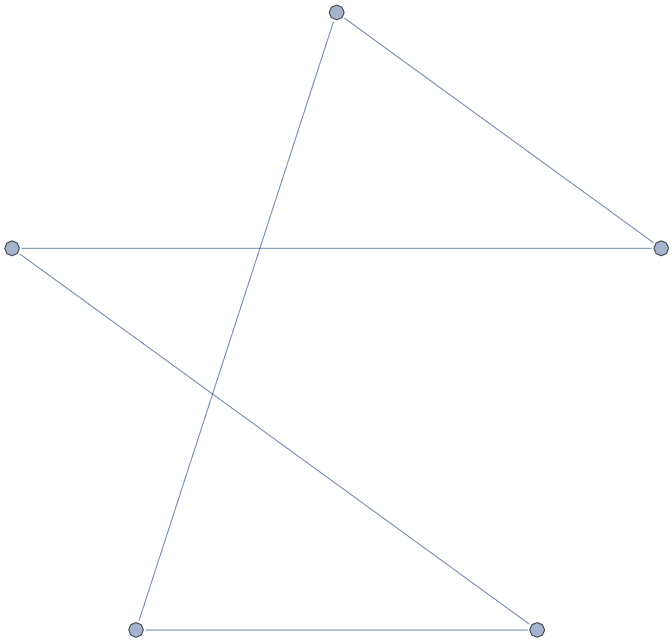
```
g = CycleGraph[5];
adjacencyLists[g]
fromAdjacencyLists[%]
h = Graph[%]
```

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

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



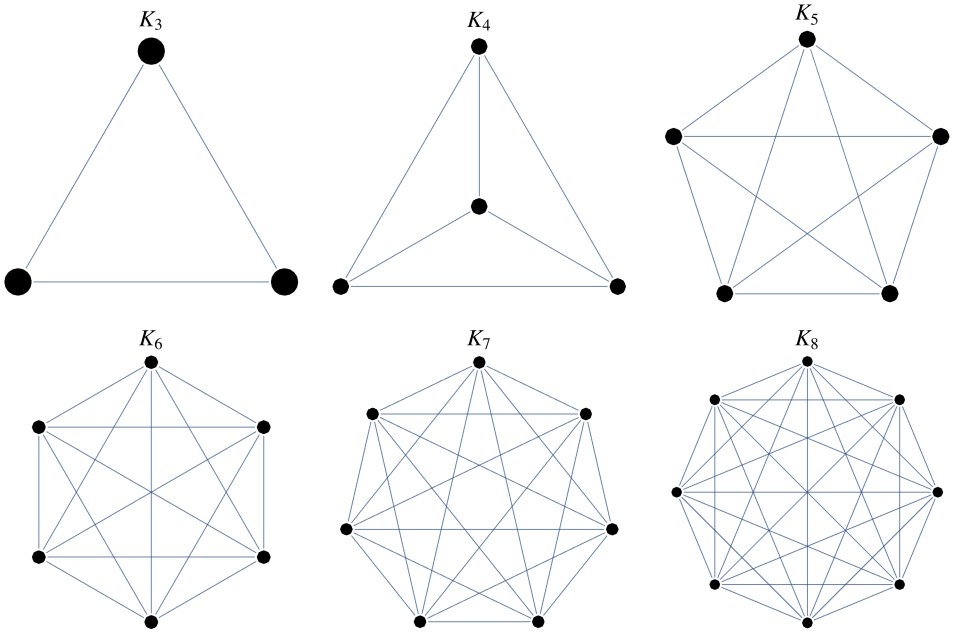
```
realizeDegreeSequence[degreeSequence[h]]
```



Special Families of Graphs

Definition 3.1. A **complete graph** is a simple graph in which every pair of distinct vertices is joined by an edge. The complete graph of n vertices is denoted by K_n.

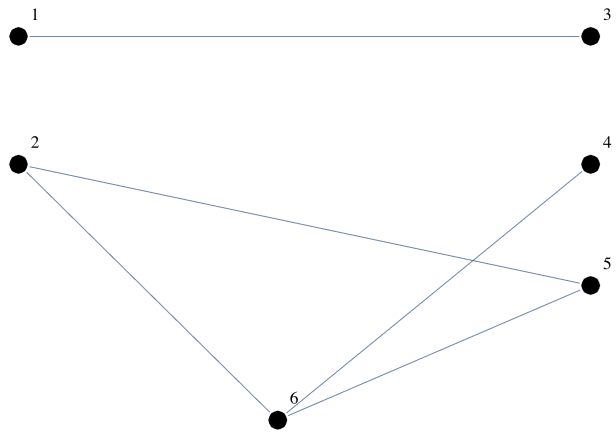
```
GraphicsGrid[Partition[
  Table[CompleteGraph[i, PlotLabel → Kᵢ, VertexStyle → Black, VertexSize → Small],
   {i, 3, 9}], 3], ImageSize → 500]
```



Definition 3.2. Let $k \geq 2$ be an integer. A graph $G = (V, E)$ is called ***k*-partite** if V admits a partiton into k sets such that every edges has its ends in different sets: vertices in the same partition set must not be adjacent. It is common to call a 2-partite graph a **bipartite graph**.

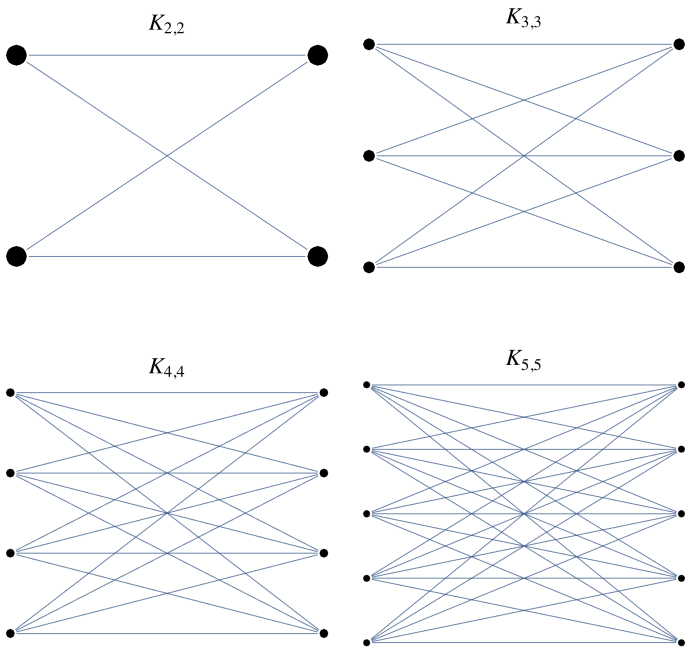
For instance, the following is a 3-partite graph.

```
Graph[Range[6], {1 ↔ 3, 2 ↔ 5, 2 ↔ 6, 4 ↔ 6, 5 ↔ 6},
 VertexCoordinates → {{-0.506, 0.37}, {-0.506, 0.18},
   {0.346, 0.37}, {0.346, 0.18}, {0.346, 0.}, {-0.12, -0.2}},
 VertexLabels → "Name", VertexSize → 0.15, VertexStyle → Black, ImagePadding → 20]
```

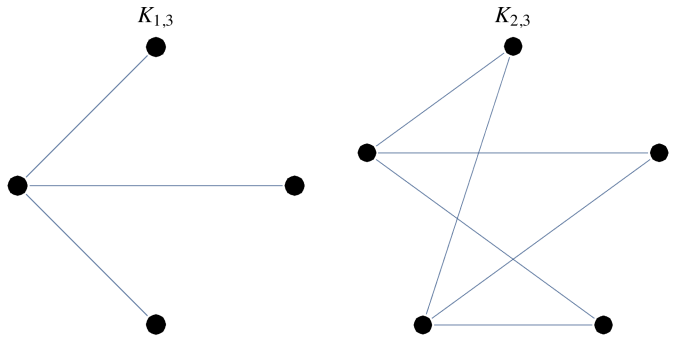


Definition 3.3. An k-partite graph in which every two vertices from different partition sets are adjacent is called **complete**. A complete k-partite graph G with partitions $G = X_1 \bigcup X_2 \bigcup \cdots \bigcup X_k$, in which each X_i has n_i vertices, is denoted by $K_{n_1, n_2, \ldots n_k}$.

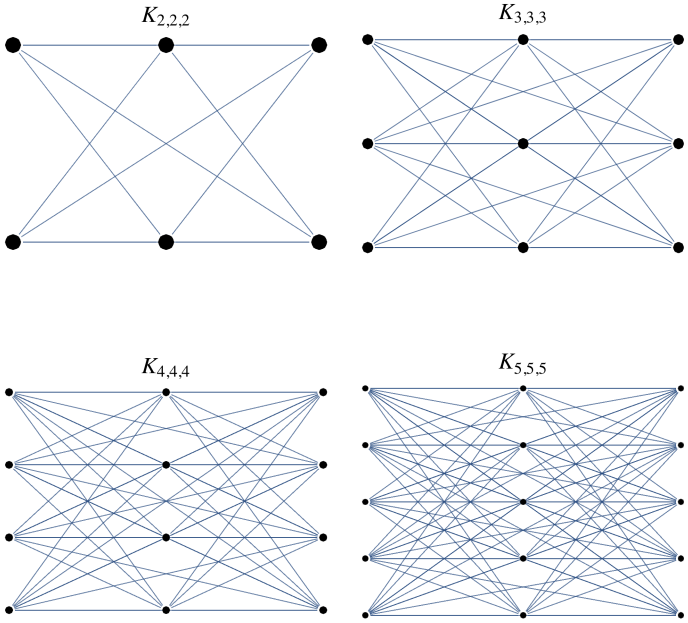
```
GraphicsGrid[Partition[Table[CompleteGraph[{i, i},
    PlotLabel → Ki,i, VertexStyle → Black, VertexSize → Small], {i, 2, 5}], 2]]
```



```
GraphicsGrid[{{CompleteGraph[{1, 3}, GraphLayout → "CircularEmbedding",
     VertexSize → Small, VertexStyle → Black, PlotLabel → K1,3],
    CompleteGraph[{2, 3}, GraphLayout → "CircularEmbedding",
     VertexSize → Small, VertexStyle → Black, PlotLabel → K2,3]}}]
```

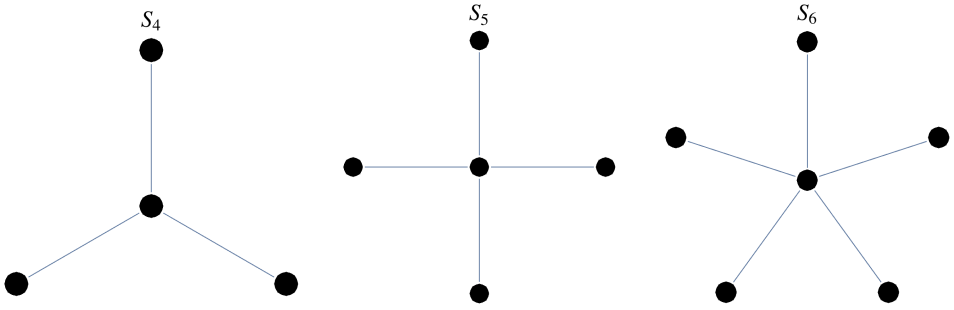



```
GraphicsGrid[Partition[Table[CompleteGraph[{i, i, i},
     PlotLabel → Ki,i,i, VertexStyle → Black, VertexSize → Small], {i, 2, 5}], 2]]
```



Definition 3.4. The complete bipartite graph $K_{1,n}$ is called the **Star graph**, denoted by S_n.

```
GraphicsGrid[
 {Table[StarGraph[i, PlotLabel → Si, VertexStyle → Black, VertexSize → 0.15], {i, 4, 6}]},
 ImageSize → 500]
```

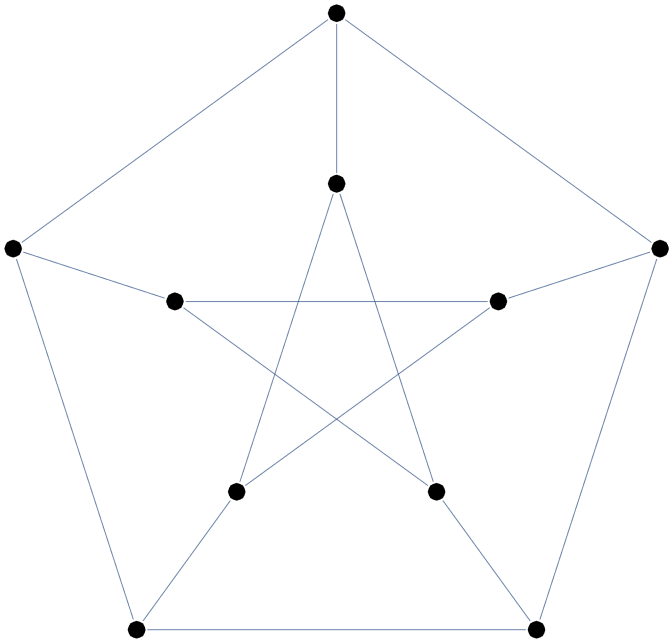


Definition 3.5. Let n, k be positive integers, where $n \geq 2\,k$. The **Kneser graph** $K(n, k)$ is the graph whose vertices correspnd to the k-element subsets of a set of n elements, and where two vertices are adjacent if the two corresponding sets are disjont.

From the definition, we know that the Kneser graph $K(n, k)$ has $\binom{n}{k}$ vertices, and each vertex has exactly $\binom{n-k}{k}$ neighbors. Therefore, $K(n, k)$ has $\binom{n}{k}\binom{n-k}{k}\Big/2$ edges. Further, $K(n, 1)$ is the complete graph K_n, and $K(n, 2)$ is the complement of the line graph of K_n.

The most famous Kneser graph is the $K(5, 2)$, which is called the **Petersen Graph**.

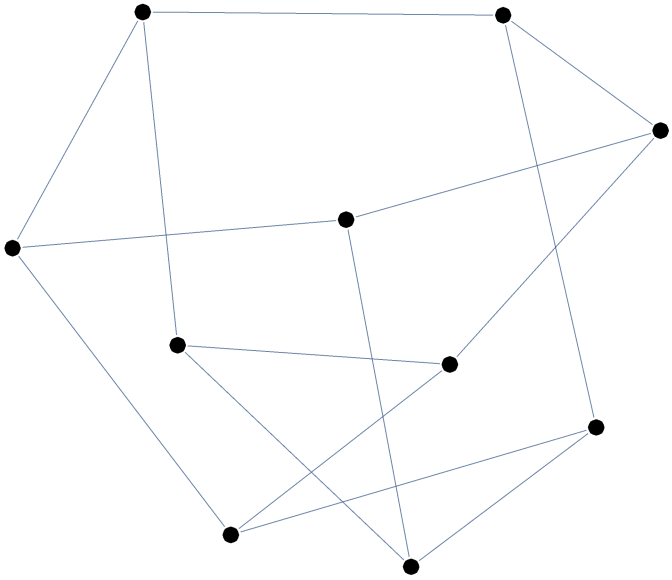
```
PetersenGraph[5, 2, VertexStyle → Black, VertexSize → Small]
```



It is easy to create the Kneser graph directly using the definition. For instance, in the following we make the Petersen graph from the definiton of $K(5, 2)$.

```
KneserGraph[{n_, k_}] :=
 Module[{vertices, edges},
  vertices = Subsets[Range[n], {k}];
  edges = Select[Subsets[vertices, {2}], Intersection[#[[1]], #[[2]]] == {} &] /.
    {x_, y_} → UndirectedEdge[x, y];
  Graph[vertices, edges, VertexSize → Small, VertexStyle → Black]
 ]
```

```
KneserGraph[{5, 2}]
```



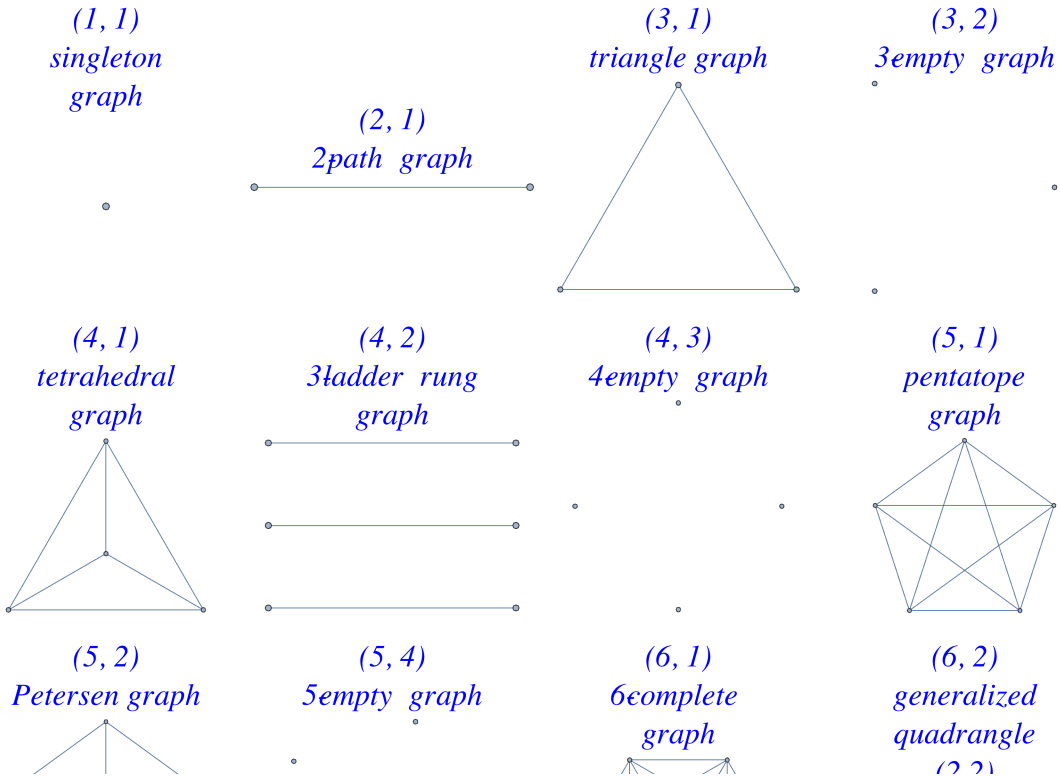
Although it does not look like the Petersen graph, it is indeed isomorprhic to the Petersen graph.

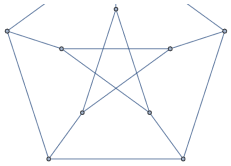
```
IsomorphicGraphQ[%, PetersenGraph[5, 2]]
```

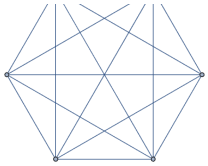
```
True
```

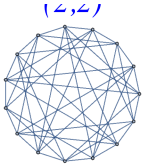
The following graphs are pulled from Wolfram server.

```
(* The following codes are extracted from mathword.wolfram.com *)
l = Sort[{"Kneser" /. GraphData[#, "NotationRules"], #} & /@
    Select[GraphData["Kneser"], GraphData[#, "Embeddings"] =!= {} &]];
GraphicsGrid[Partition[Show[GraphData[#[[1]]],
      PlotLabel → TextCell[StringReplace[ToString[#[[2]]], {"{" → "(", "}" → ")"}] <>
          "\n" <> GraphData[#[[1]], "Name"], TextAlignment → Center, PageWidth → 100],
      BaseStyle → {FontSlant → "Italic"}] & /@ Reverse /@ Take[l, 24],
  4, 4, {1, 1}, {}], ImageSize → 600, Spacings → {-10, 30}]
```



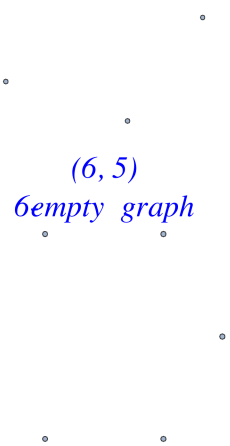




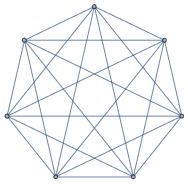


(6, 3)
10ladder
rung graph

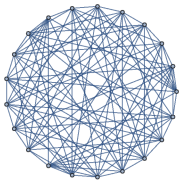
(6, 5)
6empty graph



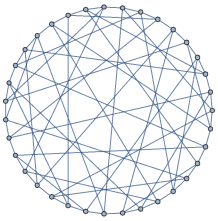
(7, 1)
7complete
graph



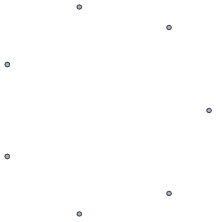
(7, 2)
(7,2)Kneser
graph



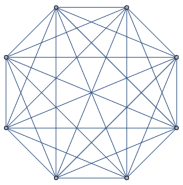
(7, 3)
4odd graph



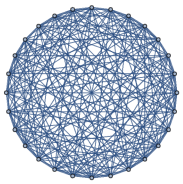
(7, 6)
7empty graph



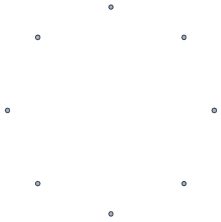
(8, 1)
8complete
graph



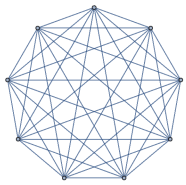
(8, 2)
(8,2)Kneser
graph



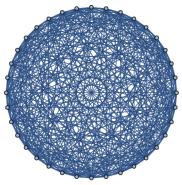
(8, 7)
8empty graph



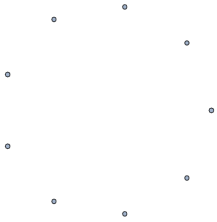
(9, 1)
9complete
graph



(9, 2)
(9,2)Kneser
graph



(9, 8)
9empty graph



Representations of Graphs

Undirected Graphs

Definition 4.1. Given a graph G with vertices indexed as $V(G) = \{v_1, \ldots, v_n\}$, the ***adjacency matrix*** of G is the $n \times n$ matrix $A(G) = (a_{ij})$, where the entry a_{ij} is the number of edges joing the vertex v_i to v_j.

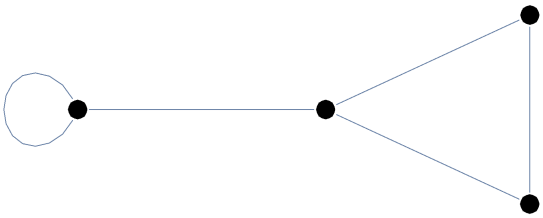
```
Grid[{{g = Graph[{"w" ↔ "x", "x" ↔ "y", "y" ↔ "z"},
     VertexCoordinates → {{0, 0}, {0, -1}, {2, 0}, {2, -1}}, VertexLabels → "Name",
     VertexSize → Small, VertexStyle → Black, ImagePadding → 20],
   TableForm[Normal@AdjacencyMatrix[g], TableHeadings →
     {Style[#, Red] & /@ VertexList[g], Style[#, Red] & /@ VertexList[g]}],
   MatrixForm[AdjacencyMatrix[g]]}}, Spacings → 4]
```

	w	x	y	z
w	0	1	0	0
x	1	0	1	0
y	0	1	0	1
z	0	0	1	0

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

Note that $A(G)$ is a symmetric matrix. If G has no loops then all the entries of the main diagonal of $A(G)$ are 0.

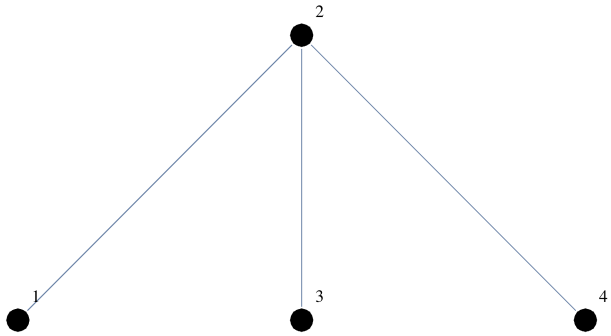
```
Grid[{{g = Graph[{1 ↔ 1, 1 ↔ 2, 2 ↔ 3, 2 ↔ 4, 3 ↔ 4},
      VertexSize → Small, VertexStyle → Black, ImageSize → 300],
   AdjacencyMatrix[g] // MatrixForm}}, Spacings → 4]
```



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

A graph can also be specficied by giving its adjacency matrix.

```
AdjacencyGraph[{{0, 1, 0, 0}, {1, 0, 1, 1}, {0, 1, 0, 0}, {0, 1, 0, 0}}, VertexLabels → "Name",
 VertexSize → 0.08, VertexStyle → Black, ImagePadding → 20]
```



Theorem 4.1. *Let $G = (V, E)$ be a graph, and let $V = \{v_1, \ldots, v_n\}$. Let k be any positive integer.*

1. The (i, j)-th entry of A^k is the number of different $v_i - v_j$ walks in G of length k.

2. Let

$$B = A + A^2 + \cdots + A^{n-1}$$

G is connected if and only if for every $i \neq j$ we have $b_{ij} \neq 0$.

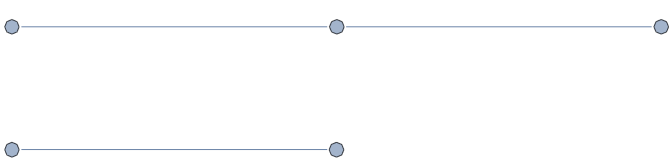
Example 3.1. Test the connectivity of the graph defined by the adjacency matrix.

```
A = {{0, 1, 0, 0, 0}, {1, 0, 0, 0, 1}, {0, 0, 0, 1, 0}, {0, 0, 1, 0, 0}, {0, 1, 0, 0, 0}};
```

```
ConnectedGraphQ[AdjacencyGraph[A]]
```

```
False
```

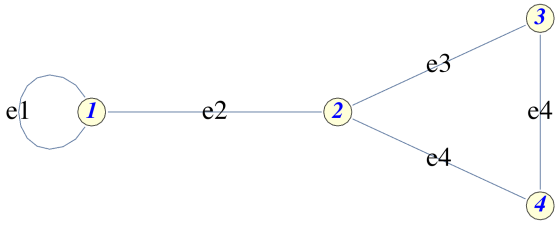
```
AdjacencyGraph[A]
```



Definition 4.2. Let $G = (V, E)$ be a graph, where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_k\}$. The ***incidence matrix*** of G is the $n \times k$ matrix $M(G) = (m_{ij})$ where m_{ij} is defined by

$$m_{ij} = \begin{cases} 0 & v_i \text{ is not an end of } e_j \\ 1 & v_i \text{ is an end of the nonloop } e_j \\ 2 & v_i \text{ is an end of the loop } e_j \end{cases}$$

```
Grid[{{g = Graph[{Labeled[1 ⟷ 1, "e1"], Labeled[1 ⟷ 2, "e2"], Labeled[2 ⟷ 3, "e3"],
       Labeled[2 ⟷ 4, "e4"], Labeled[3 ⟷ 4, "e4"]}, VertexStyle → LightYellow,
      VertexSize → 0.15, VertexLabels → Placed["Name", {1 / 2, 1 / 2}],
      VertexLabelStyle → Directive[12, Blue, Bold, Italic],
      EdgeLabelStyle → Directive[14, Black], ImageSize → 300],
     IncidenceMatrix[g] // MatrixForm}}, Spacings → 4]
```



$$\begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

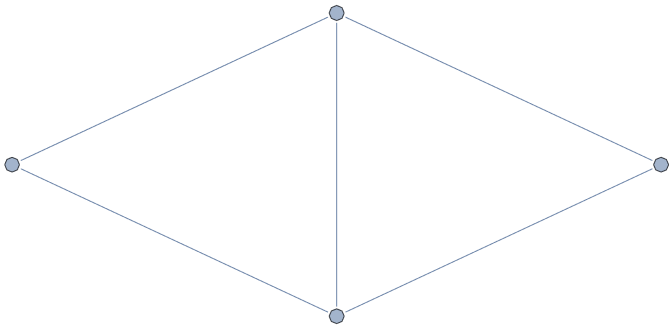
```
TableForm[Normal@IncidenceMatrix[g],
 TableHeadings → {Style[#, Red] & /@ VertexList[g], Style[#, Red] & /@ EdgeList[g]}]
```

	1 ⟷ 1	1 ⟷ 2	2 ⟷ 3	2 ⟷ 4	3 ⟷ 4
1	2	1	0	0	0
2	0	1	1	1	0
3	0	0	1	0	1
4	0	0	0	1	1

Notice that the sum of the elements in the ith row of $M(G)$ gives us the degree of the vertex v_i, while the sum of elements in each column is 2 (corresponding to the 2 ends of the edges).

We can also obtain the graph from its incidence matrix.

```
IncidenceGraph[{{1, 1, 0, 0, 0}, {1, 0, 1, 1, 0}, {0, 1, 1, 0, 1}, {0, 0, 0, 1, 1}}]
```



Circulant Graphs

In this section, all graphs are simple.

Definition 5.1. An $n \times n$ matrix $A = (a_{ij})$ is said to be a **circulant matrix** if its entries satisfies $a_{i,j} = a_{1,j-i+1}$, where the subscripts are reduced modulo n and lie in the set $\{1, 2, \ldots, n\}$.

From the definition, we see

$$\begin{array}{lll} a_{1,1} = a_{1,1} & a_{1,2} = a_{1,2} & \dots \quad a_{1,n} = a_{1,n} \\ a_{2,1} = a_{1,1-2+1} = a_{1,n} & a_{2,2} = a_{1,2-2+1} = a_{1,1} & \dots \quad a_{2,n} = a_{1,n-2+1} = a_{1,n-1} \\ a_{3,1} = a_{1,1-3+1} = a_{1,n-1} & a_{3,2} = a_{1,2-3+1} = a_{1,n} & \dots \quad a_{3,n} = a_{1,n-3+1} = a_{1,n-2} \end{array}$$

Therefore, the entries of A are determined by the entries on the first row of A. Further, starting at the second row, each row is a right rotation of the previous row. For example, if A is a 3×3 circulant matrix, then

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{1,3} & a_{1,1} & a_{1,2} \\ a_{1,2} & a_{1,3} & a_{1,1} \end{pmatrix}.$$

Notice that

$$A = a_{1,1}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + a_{1,2}\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} + a_{1,3}\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

and that these three matrices are themselves ciculant matrices. Since

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}^0 \quad \text{and} \quad \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}^2,$$

all 3×3 circulant matrices are generated by

$$W_3 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix},$$

which is itself generated by the first row $(0, 1, 0)$.

Consequently, for each $n \geq 2$, we define W_n to be the ciculant matrix generated by the vector $(0, 1, 0, \dots, 0)$, and we call

$$\{I_n, W_n, {W_n}^2, \dots, {W_n}^{n-1}\}$$

the **circulant basis**. It is easy to verify that ${W_n}^j, 0 \leq j \leq n-1$, are circulant matrices and that if A is an $n \times n$ circulant matrix, with first row $(a_1, a_2, \dots, a_n)$, then

$$A = a_1 I_n + a_2 W_n + a_3 {W_n}^2 + \cdots + a_n {W_n}^{n-1}.$$

```
circulantMatrixBase[n_ /; n ≥ 2] :=
 Module[{u, i},
  u = Fold[Append, {0, 1}, Table[0, {i, n - 2}]];
  Fold[Join, {u}, Table[{u = RotateRight[u]}, {i, n - 1}]]
 ]
```

■ **Example 5.1:** 4×4 circulant matrix

```
n = 4;
W = circulantMatrixBase[n];
base = Table[MatrixPower[W, k], {k, 0, n - 1}];
MatrixForm /@ base
```

$$\sum_{k=1}^{n} a_k \, \text{base}[[k]] \;//\; \text{MatrixForm}$$

$$\left\{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}\right\}$$

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_4 & a_1 & a_2 & a_3 \\ a_3 & a_4 & a_1 & a_2 \\ a_2 & a_3 & a_4 & a_1 \end{pmatrix}$$

Definition 5.2. A graph $G = (V, E)$ is **circulant**, if V can be obtained so that the adjacency matrix $A(G)$ is a circulant matrix.

Recall that the adjacency matrix of a graph is symmetric and its diagonal entries are zero. Therefore, not all circulant matrices can be the adjacency matrix of some graphs. Lets find out the necessary and sufficient condition that a circulant matrix is also the adjacency matrix of some circulant graph.

If G is circulant, then $A(G)$ is circulant. Thus, $A(G)$ can be determined by its first row. Since the entries of the main diagonal of $A(G)$ are zero, the first row of $A(G)$ must be of the form

$$(0, a_2, \ldots, a_n).$$

However, since $A(G)$ is symmetric, if

$$A(G) = \begin{pmatrix} 0 & a_2 & a_3 & a_4 & \ldots & a_n \\ a_n & 0 & a_2 & a_3 & \ldots & a_{n-1} \\ a_{n-1} & a_n & 0 & a_2 & \ldots & a_{n-2} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{pmatrix}$$

then

$$a_2 = a_n,\ a_3 = a_{n-1}, a_4 = a_{n-2}, \ldots$$

Hence, $a_2, a_3, a_4, \ldots, a_n$ have to satisfy the condition

$$a_i = a_{n-i+2}, \quad i = 2, 3, \ldots, n.$$

So, according to the parity of n, to obtain the adjacency matrix $A(G)$ we only need to provide by the following information.

$$\begin{cases} a_2, a_3, \ldots, a_{\lceil n/2 \rceil}, & n \text{ is odd} \\ a_2, a_3, \ldots, a_{n/2+1}, & n \text{ is even} \end{cases}$$

For instance, when $n = 2, 4, 6, 8$, the first row is given by

n	specified entries	First row
2	a_2	$(0, a_2)$
4	a_2, a_3	$(0, a_2, a_3, a_2)$
6	a_2, a_3, a_4	$(0, a_2, a_3, a_4, a_3, a_2)$
8	a_2, a_3, a_4, a_5	$(0, a_2, a_3, a_4, a_5, a_4, a_3, a_2)$

Similarly, when $n = 3, 5, 7, 9$, we get the following table.

n	specified entries	First row
3	a_2	$(0, a_2, a_2)$
5	a_2, a_3	$(0, a_2, a_3, a_3, a_2)$
7	a_2, a_3, a_4	$(0, a_2, a_3, a_4, a_4, a_3, a_2)$

9 a_2, a_3, a_4, a_5 $\quad$ $(0, a_2, a_3, a_4, a_5, a_5, a_4, a_3, a_2)$

Observe that, no matter n is even or odd, both tables require the same number of information. For instance, when $n = 2$ or $n = 3$, we need to specify the entry a_2; when $n = 3, 4$, we need to specify entries a_2 and a_3. Therefore, in order to obtain the adjacency matrix $A(G)$, it is not enough to just simply provide the reqired entries. We have to give one addition information: whether n is even or odd.

```
circulantAdjMatrix[lst_, option_: Even] :=
 Module[{m = Length[lst], n, firstRow, A, k},
  firstRow = Join[{0}, lst];
  If[option === Even,
   n = 2 m;
   firstRow = Join[firstRow, Reverse[Drop[lst, -1]]],
   n = 2 m + 1;
   firstRow = Join[firstRow, Reverse[lst]]
  ];
  A = circulantMatrixBase[n];
  Dot[firstRow, Table[MatrixPower[A, k], {k, 0, n - 1}]]
 ]
```

We now write a program to generate all adjacency matrices of the circulant graph of given order.

```
circulantAdjMatList[n_ /; n ≥ 2] :=
 Module[{zeroOneCombinations, adjMat},
  zeroOneCombinations = Tuples[{0, 1}, Floor[n / 2]];
  If[EvenQ[n],
   adjMat = circulantAdjMatrix[#] & /@ zeroOneCombinations,
   adjMat = circulantAdjMatrix[#, Odd] & /@ zeroOneCombinations
  ];
  adjMat
 ]
```

■ **Example 5.2:** The adjacency matrices of circulant graphs of order 2 and 3.

```
MatrixForm /@ circulantAdjMatList[2]
MatrixForm /@ circulantAdjMatList[3]
```

$$\left\{\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\right\}$$

$$\left\{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}\right\}$$

■ **Example 5.3:** The adjacency matrices of circulant graphs of order 4 and 5.

```
MatrixForm /@ circulantAdjMatList[4]
MatrixForm /@ circulantAdjMatList[5]
```

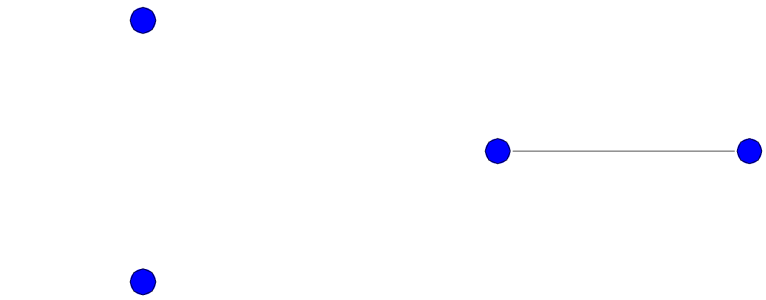
$$\left\{\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}\right\}$$

$$\left\{\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}\right\}$$

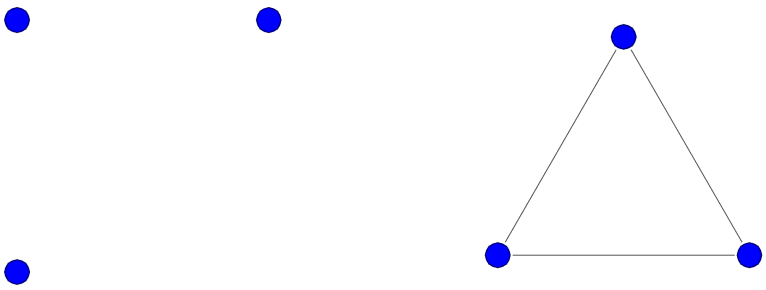
In the following, we list all circulant graph of order up to 9.

```
circulantGraphList[n_ /; n ≥ 2, {grpOpts___}, grpGridOpts___] :=
 Module[{adjMList, gList},
  adjMList = circulantAdjMatList[n];
  gList = AdjacencyGraph[#, grpOpts] & /@ adjMList;
  If[Length[gList] > 4,
   GraphicsGrid[Partition[gList, 4], grpGridOpts],
   GraphicsGrid[{gList}, grpGridOpts]]
 ]
```

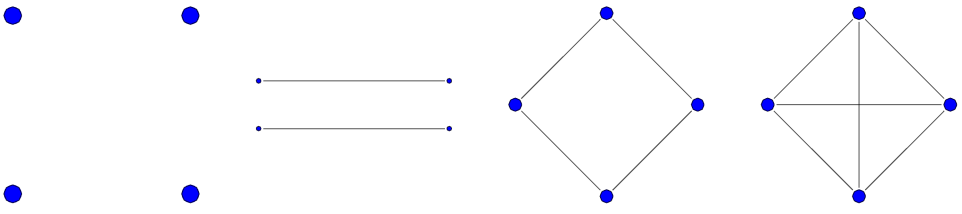
```
circulantGraphList[2, {VertexStyle → Blue, EdgeStyle → Black, VertexSize → 0.1},
 ImageSize → 400, Spacings → Scaled[1]]
```



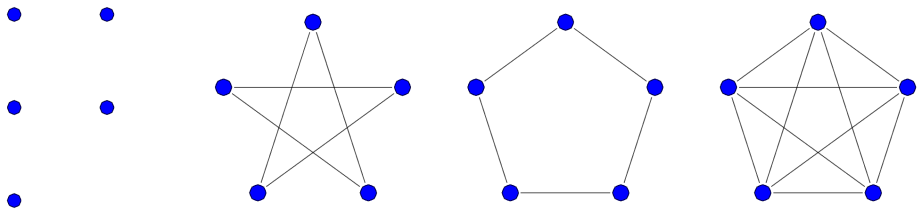
```
circulantGraphList[3, {VertexStyle → Blue, EdgeStyle → Black, VertexSize → 0.1},
 ImageSize → 400, Spacings → Scaled[1]]
```



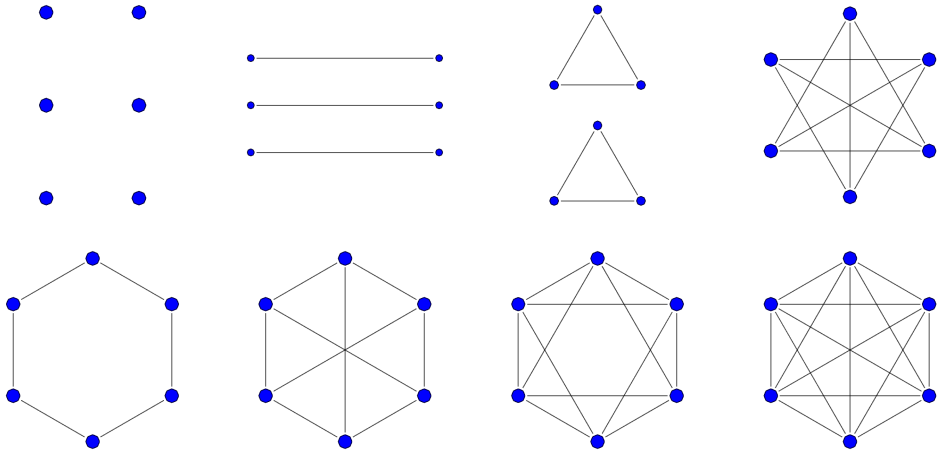
```
circulantGraphList[4, {VertexStyle → Blue, EdgeStyle → Black, VertexSize → 0.1},
 ImageSize → 500, Spacings → Scaled[.3]]
```



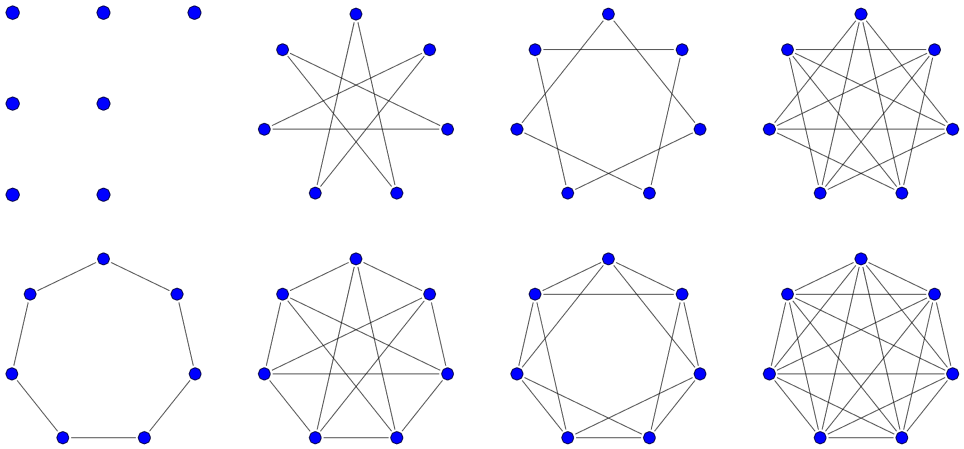
```
circulantGraphList[5, {VertexStyle → Blue, EdgeStyle → Black, VertexSize → 0.15},
 ImageSize → 500, Spacings → Scaled[.3]]
```



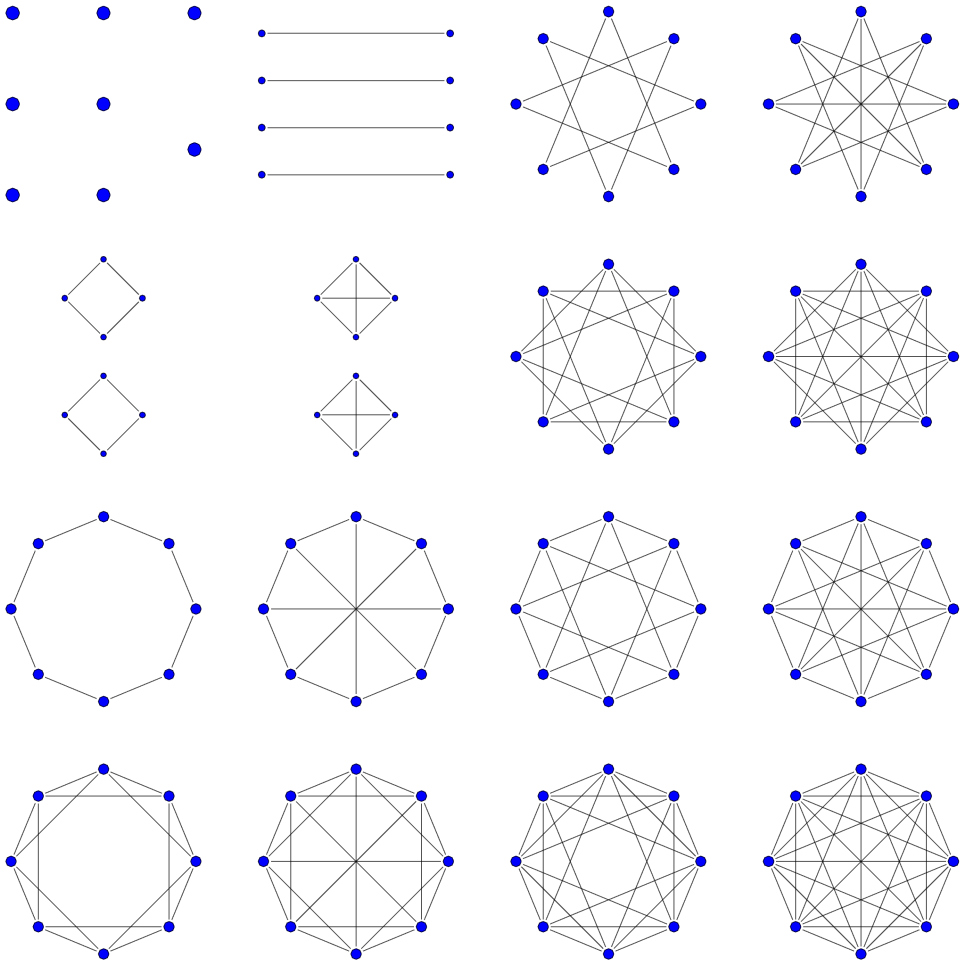
```
circulantGraphList[6, {VertexStyle → Blue, VertexSize → 0.15, EdgeStyle → Black,
  GraphLayout → "CircularEmbedding"}, ImageSize → 500, Spacings → Scaled[.3]]
```



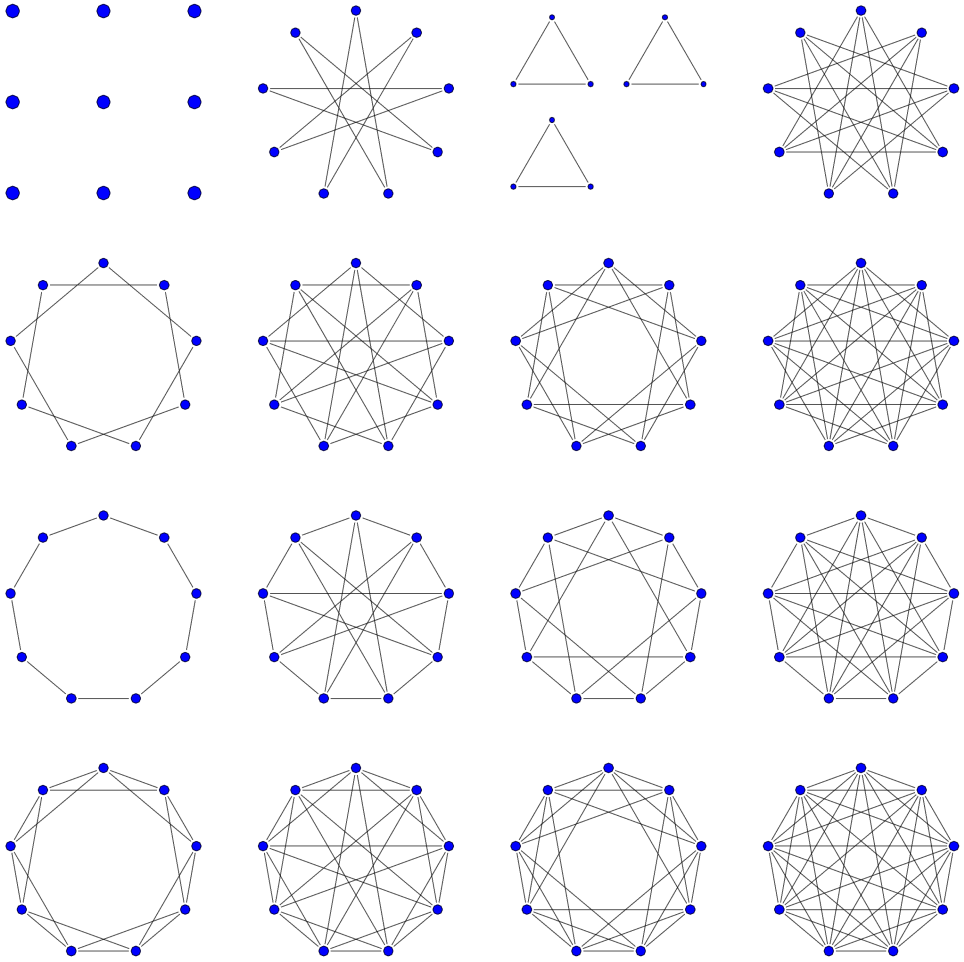
```
circulantGraphList[7, {VertexStyle → Blue, EdgeStyle → Black, VertexSize → 0.15,
  GraphLayout → "CircularEmbedding"}, ImageSize → 500, Spacings → Scaled[.3]]
```



```
circulantGraphList[8, {VertexStyle → Blue, EdgeStyle → Black, VertexSize → 0.15,
  GraphLayout → "CircularEmbedding"}, ImageSize → 500, Spacings → Scaled[.3]]
```

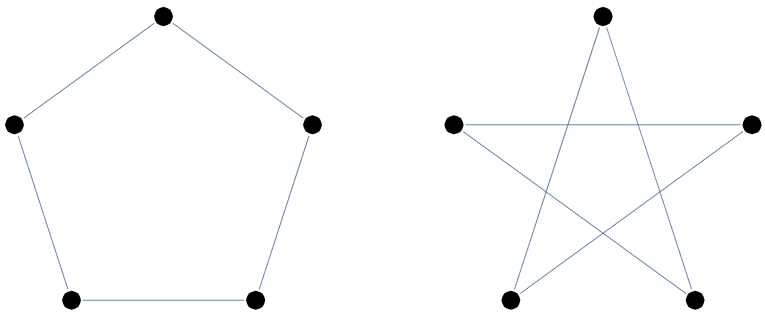


```
circulantGraphList[9, {VertexStyle → Blue, EdgeStyle → Black, VertexSize → 0.15,
  GraphLayout → "CircularEmbedding"}, ImageSize → 500, Spacings → Scaled[.3]]
```



Although the program we wrote above can give us all the circulant graphs of a given order, there is one thing missing from the picture. That is, we did not consider the isomorphic problem. There are many graphs that are actually isomorphic but still coexistent in the graphs list. For example, the following are isomorphic circulant graphs, and they are both in the list.

```
GraphicsGrid[{{CirculantGraph[5, 1, VertexStyle → Black, VertexSize → Small],
    CirculantGraph[5, 2, VertexStyle → Black, VertexSize → Small]}},
 Spacings → Scaled[0.5], ImageSize → 400]
```

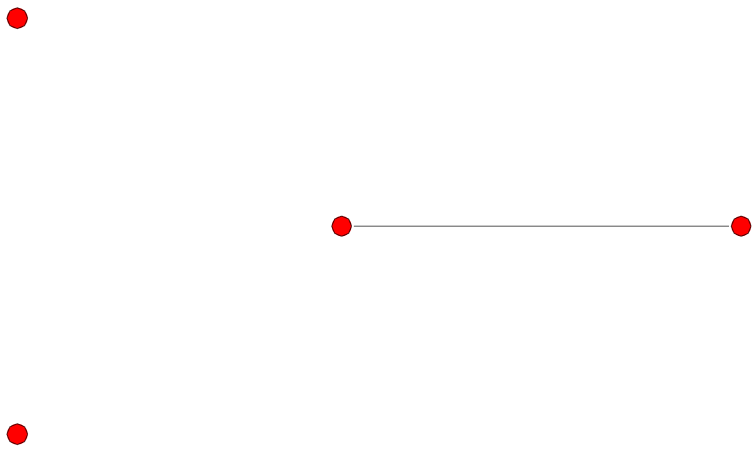


The following is the modified program that takes care the isomorphic problem.

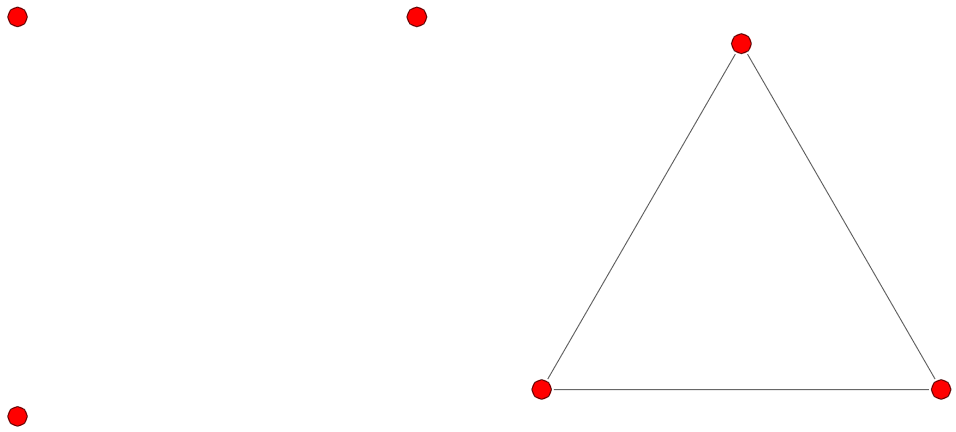
```
circulantGraphIsoList[n_ /; n ≥ 2, {grpOpts___}, grpGridOpts___] :=
 Module[{gList, m, nonIsoList = {}, i, j, g, num},
  m = 2^Floor[n/2];
  gList = AdjacencyGraph[#, grpOpts] & /@ circulantAdjMatList[n];
  nonIsoList = Append[nonIsoList, gList[[1]]];
  For[i = 2, i ≤ m, i++,
   g = gList[[i]];
   num = Length[nonIsoList];
   j = 1;
   While[j ≤ num && Not[IsomorphicGraphQ[nonIsoList[[j]], g]], j++];
   If[j > num, nonIsoList = Append[nonIsoList, g]];
  ];
  If[Length[nonIsoList] > 4,
   GraphicsGrid[Partition[nonIsoList, 4], grpGridOpts],
   GraphicsGrid[{nonIsoList}, grpGridOpts]]
]
```

```
Do[
 Print["nonisomorphic circulant graphs of order ", n];
 Print["------------------------------------------------------------------------"];
 If[n ≥ 4,
  Print[
   circulantGraphIsoList[n, {VertexStyle → Red, EdgeStyle → Black, VertexSize → 0.15,
     GraphLayout → "CircularEmbedding"}, ImageSize → 500, Spacings → Scaled[.3]]],
  Print[circulantGraphIsoList[n, {VertexStyle → Red, EdgeStyle → Black,
     VertexSize → 0.05, GraphLayout → "CircularEmbedding"},
    ImageSize → 500, Spacings → Scaled[.3]]]
 ],
 {n, 2, 8}]
```

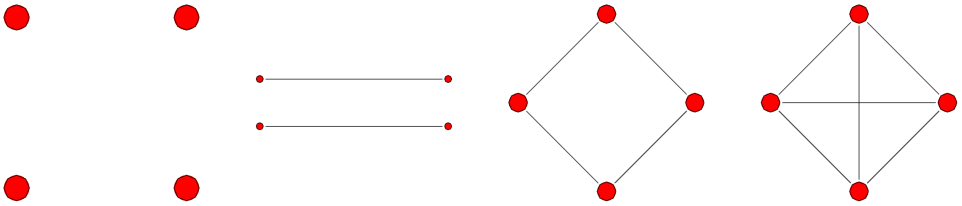
```
nonisomorphic circulant graphs of order 2

------------------------------------------------------------------------
```



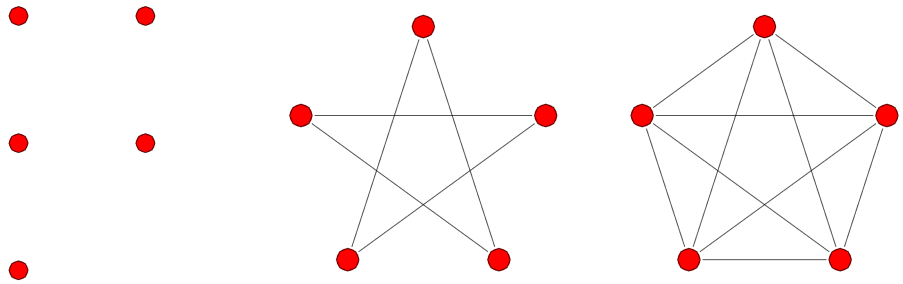
```
nonisomorphic circulant graphs of order 3

------------------------------------------------------------------------
```



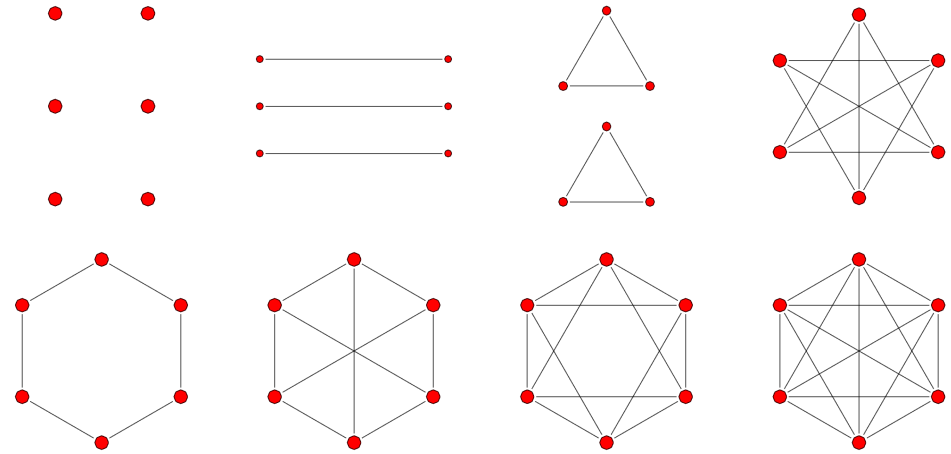
nonisomorphic circulant graphs of order 4



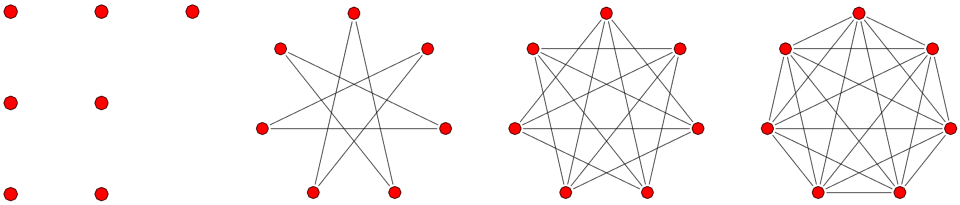
nonisomorphic circulant graphs of order 5



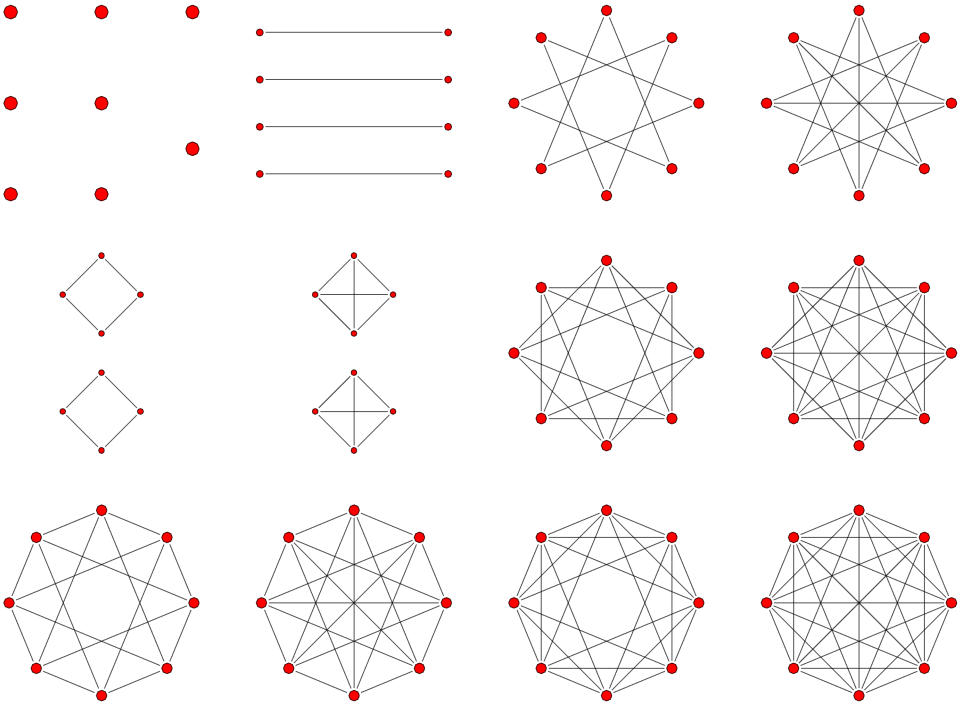
nonisomorphic circulant graphs of order 6



nonisomorphic circulant graphs of order 7



nonisomorphic circulant graphs of order 8



Graph Traversals

Breadth First Search

Breadth-First Search Algorithm
Input: A graph $G=(V, E)$ of order $n>0$. A vertex s from which to start the search. The

vertices are numbered from 1 to n, i.e., $V = \{1, 2, \ldots, n\}$.
Output: A list D of distances of all vertices from s. A tree T rooted at s.

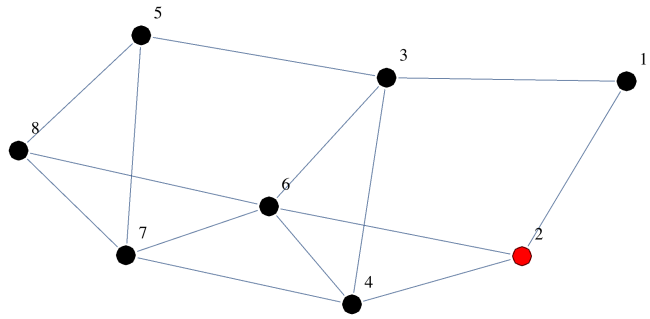
```
 1: Q ← {s}, where Q is the queue of nodes to visit.
 2: D ← {∞, ∞, …, ∞}, n copies of ∞.
 3: D[s] ← 0
 4: T ← {}
 5: While Q ≠ {} do
 6:     v ← dequeue Q
 7:     for each w ∈ adj(v) do
 8:         if D[w] = ∞, then
 9:             D[w] ← D[v] + 1
10:             enqueue (Q, w)
11:             append (T, v w)
12: return (D, T)
```

The operation of removing the front of Q is referred to as dequeue, while the operation of appending to the rear of Q is called enqueue.

```
(* Breadth-First Search *)
BFS[g_Graph, start_Integer] :=
 Module[{e, v, cnt = 1, queue = {start},
   visit = Table[0, {VertexCount[g]}],   (* visiting indices *)
   parent = Table[i, {i, VertexCount[g]}], (* the parent of each vertex *)
   dist = Table[∞, {VertexCount[g]}]
   (* distance of each vertex from start *)
  },
  e = AdjacencyList[g, #] & /@ VertexList[g];
  visit[[start]] = cnt++;
  dist[[start]] = 0;
  While[queue ≠ {},
   {v, queue} = {First[queue], Rest[queue]};
   Scan[(If[visit[[#]] == 0, visit[[#]] = cnt++;
        parent[[#]] = v; dist[[#]] = dist[[v]] + 1; AppendTo[queue, #]]) &, e[[v]]];
  ];
  {visit, parent, dist}
 ]
```

■ **Example 6.1:** Apply the BFS algorithm to the graph shown below starting at vertex 2.

```
g = Graph[Range[8], {1 ↔ 2, 1 ↔ 3, 2 ↔ 4, 2 ↔ 6, 3 ↔ 4, 3 ↔ 5, 3 ↔ 6,
    4 ↔ 6, 4 ↔ 7, 5 ↔ 7, 5 ↔ 8, 6 ↔ 7, 6 ↔ 8, 7 ↔ 8}, VertexStyle → Black,
   VertexLabels → "Name", VertexSize → 0.15, ImagePadding → 10];
g = SetProperty[{g, 2}, VertexStyle → Red]
```



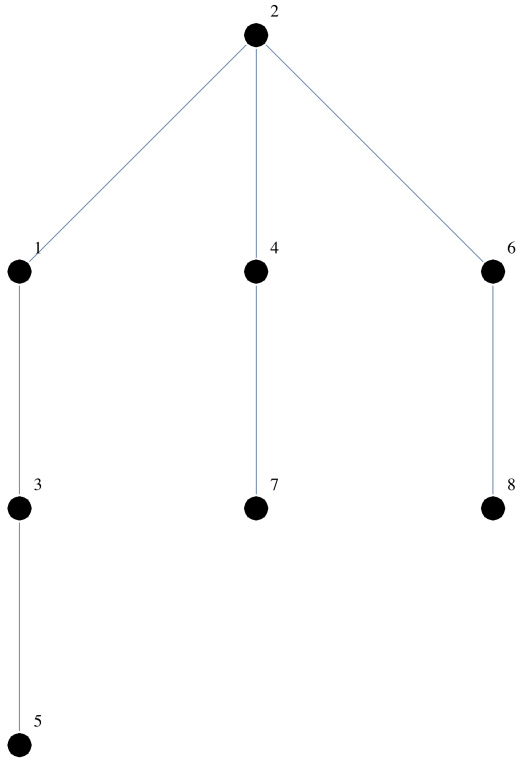

```
{visit, parent, distance} = BFS[g, 2]
```

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

```
Grid[{Prepend[Range[8], "Distance"], Prepend[distance, 2]},
 Dividers → {{2 → Red}, {2 → Red}}]
```

Distance	1	2	3	4	5	6	7	8
2	1	0	2	1	3	1	2	2

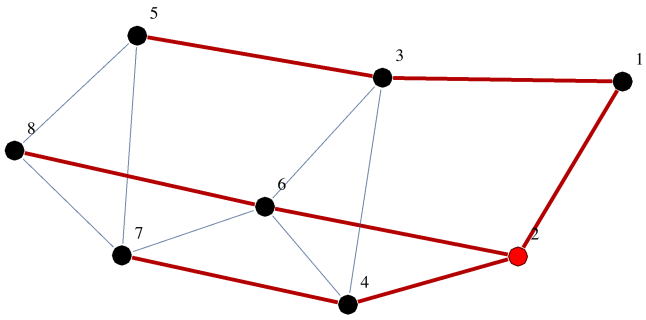
```
TreeGraph[VertexList[g], parent, VertexLabels → "Name",
 ImagePadding → 20, VertexStyle → Black, VertexSize → Small,
 VertexCoordinates → {{0.7, 1.1}, {0.8, 1.2}, {0.7, 1.0},
   {0.8, 1.1}, {0.7, 0.9}, {0.9, 1.1}, {0.8, 1.0}, {0.9, 1.0}}]
```




```
Transpose[{VertexList[g], parent}] /. {x_, y_} → UndirectedEdge[x, y]
```

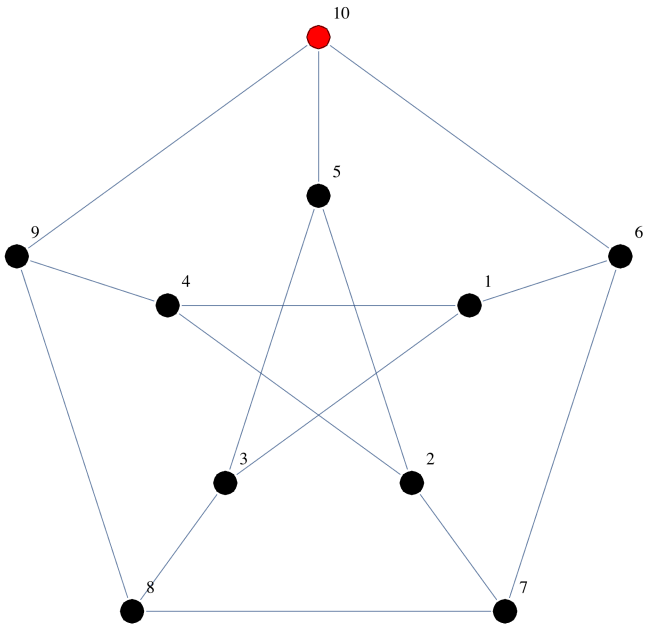
```
{1 ↔ 2, 2 ↔ 2, 3 ↔ 1, 4 ↔ 2, 5 ↔ 3, 6 ↔ 2, 7 ↔ 4, 8 ↔ 6}
```

```
HighlightGraph[g, %, GraphHighlightStyle → "Thick"]
```



■ **Example 6.2:** Apply the BFS algorithm to the Petersen graph at vertex 10.

```
g = PetersenGraph[5, 2, VertexStyle → Black,
    VertexSize → 0.15, VertexLabels → "Name", ImagePadding → 10];
g = SetProperty[{g, 10}, VertexStyle → Red]
```



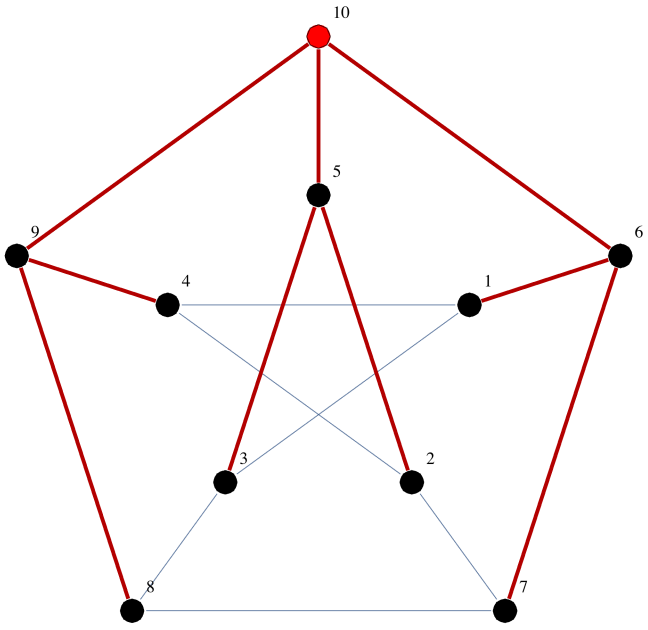

```
{visit, parent, distance} = BFS[g, 10]
```

```
{{7, 5, 6, 9, 2, 3, 8, 10, 4, 1},
 {6, 5, 5, 9, 10, 10, 6, 9, 10, 10}, {2, 2, 2, 2, 1, 1, 2, 2, 1, 0}}
```

```
Grid[{Prepend[Range[VertexCount[g]], "Distance"], Prepend[%[[3]], 10]},
 Dividers → {{2 → Red}, {2 → Red}}]
```

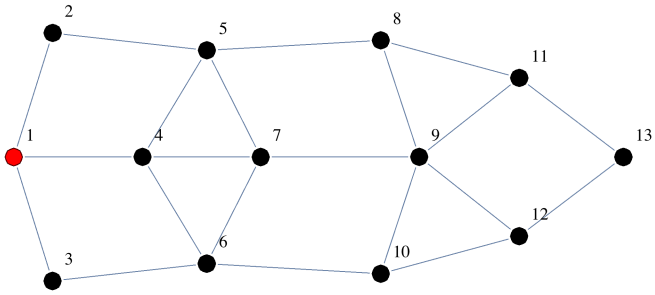
Distance	1	2	3	4	5	6	7	8	9	10
10	2	2	2	2	1	1	2	2	1	0

```
Transpose[{VertexList[g], parent}] /. {x_, y_} → UndirectedEdge[x, y];
HighlightGraph[g, %, GraphHighlightStyle → "Thick"]
```

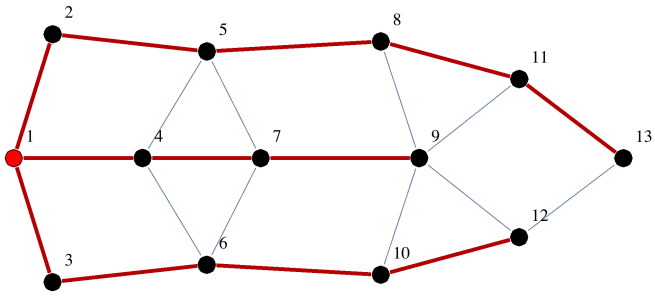


■ **Example 6.3:** `BreadthFirstScan` is a built-in *Mathematica* function that implements the Breadth-First Search Algorithm. In this example, use BreadthFirstScan to find a rooted spanning tree of the graph given by its adjacency matrix.

```
A = {{0,1,1,1,0,0,0,0,0,0,0,0,0},
     {1,0,0,0,1,0,0,0,0,0,0,0,0},
     {1,0,0,0,0,1,0,0,0,0,0,0,0},
     {1,0,0,0,1,1,1,0,0,0,0,0,0},
     {0,1,0,1,0,0,1,1,0,0,0,0,0},
     {0,0,1,1,0,0,1,0,0,1,0,0,0},
     {0,0,0,1,1,1,0,0,1,0,0,0,0},
     {0,0,0,0,1,0,0,0,1,0,1,0,0},
     {0,0,0,0,0,0,1,1,0,1,1,1,0},
     {0,0,0,0,0,1,0,0,1,0,0,1,0},
     {0,0,0,0,0,0,0,1,1,0,0,0,1},
     {0,0,0,0,0,0,0,0,1,1,0,0,1},
     {0,0,0,0,0,0,0,0,0,0,1,1,0}};
g = AdjacencyGraph[A, VertexLabels → "Name",
    VertexStyle → Black, VertexSize → 0.15, ImagePadding → 10];
g = SetProperty[{g, 1}, VertexStyle → Red]
```




```
HighlightGraph[g, Reap[BreadthFirstScan[g, 1, {"FrontierEdge" → Sow}]][[2, 1]],
 GraphHighlightStyle → "Thick"]
```



Depth First Search

Depth-First Search Algorithm
Input: A graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to n, i.e., $V = \{1, 2, \ldots, n\}$.
Output: A list D of distances of all vertices from s. A tree T rooted at s.

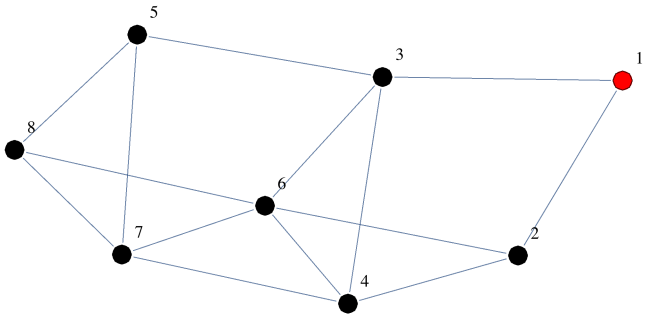
```
 1: S ← {s}, where S is the stack of nodes to visit.
 2: D ← {∞, ∞, …, ∞}, n copies of ∞.
 3: D[s] ← 0
 4: T ← {}
 5: push(S, s)
 6: While S ≠ {} do
 7:     v ← pop(S)
 8:    if unlabeled v, then labeled v
 9:       for each w ∈ adj(v) do
10:           if unlabeled w, then
11:              D[w] ← D[v] + 1
11:              push(S, w)
12:              append (T, v w)
```

13: **return** (D, T)
The operation of removing the top element of the stack is referred to as popping the element off the stack. Inserting an element into the stack is called pushing the element onto the stack.

```
DFS[g_Graph, start_Integer] := Module[
  {v, e, visited = Table[False, {VertexCount[g]}],
   parent = Table[i, {i, VertexCount[g]}],
   dist = Table[∞, {VertexCount[g]}],
   stack = {start}},
  dist[[start]] = 0;
  e = AdjacencyList[g, #] & /@ VertexList[g];
  While[stack ≠ {},
   {v, stack} = {Last[stack], Most[stack]};
   If[Not[visited[[v]]],
    visited[[v]] = True;
    Scan[(If[Not[visited[[#]]], parent[[#]] = v;
         dist[[#]] = dist[[v]] + 1; AppendTo[stack, #]]) &, e[[v]];
   ]
  ];
  {parent, dist}
 ]
```

■ **Example 6.4:** Apply the DFS algorithm to the graph shown below starting at vertex 1.

```
g = Graph[Range[8], {1 ↔ 2, 1 ↔ 3, 2 ↔ 4, 2 ↔ 6, 3 ↔ 4, 3 ↔ 5, 3 ↔ 6,
     4 ↔ 6, 4 ↔ 7, 5 ↔ 7, 5 ↔ 8, 6 ↔ 7, 6 ↔ 8, 7 ↔ 8}, VertexStyle → Black,
   VertexLabels → "Name", VertexSize → 0.15, ImagePadding → 10];
g = SetProperty[{g, 1}, VertexStyle → Red]
```



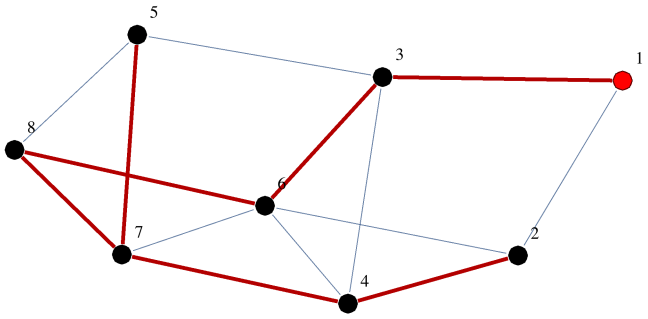

```
{parent, distance} = DFS[g, 1]
```

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

```
Grid[{Prepend[Range[VertexCount[g]], "Distance"], Prepend[distance, 1]},
 Dividers → {{2 → Red}, {2 → Red}}]
```

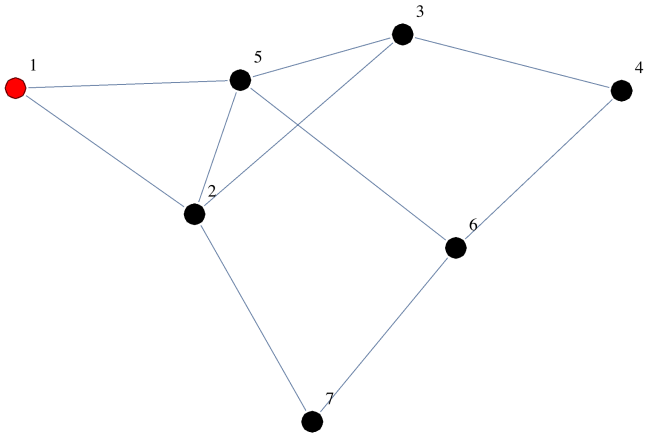
Distance	1	2	3	4	5	6	7	8
1	0	6	1	5	5	2	4	3

```
Transpose[{VertexList[g], parent}] /. {x_, y_} → UndirectedEdge[x, y];
HighlightGraph[g, %, GraphHighlightStyle → "Thick"]
```



■ **Example 6.5:** Apply the DFS algorithm to the graph shown below starting at vertex 1.

```
g = Graph[Range[7], {1 ↔ 2, 1 ↔ 5, 2 ↔ 3, 2 ↔ 5, 2 ↔ 7, 3 ↔ 4, 3 ↔ 5, 4 ↔ 6, 5 ↔ 6, 6 ↔ 7},
    VertexLabels → "Name", ImagePadding → 10, VertexStyle → Black, VertexSize → 0.15];
g = SetProperty[{g, 1}, VertexStyle → Red]
```



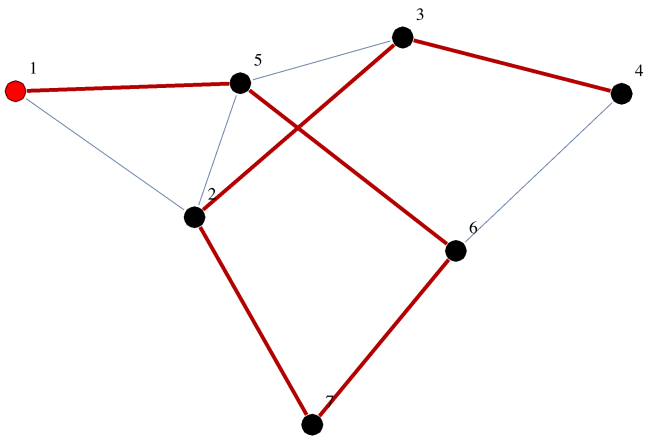

```
{parent, distance} = DFS[g, 1]
```

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

```
Grid[{Prepend[Range[VertexCount[g]], "Distance"], Prepend[distance, 1]},
 Dividers → {{2 → Red}, {2 → Red}}]
```

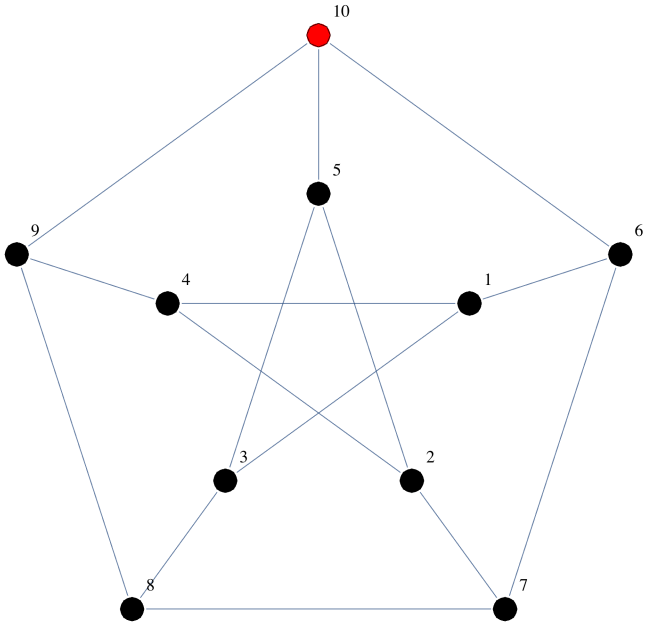
Distance	1	2	3	4	5	6	7
1	0	4	5	6	1	2	3

```
Transpose[{VertexList[g], parent}] /. {x_, y_} → UndirectedEdge[x, y];
HighlightGraph[g, %, GraphHighlightStyle → "Thick"]
```



■ **Example 6.6:** Apply the DFS algorithm to the Petersen graph at vertex 10.

```
g = PetersenGraph[5, 2, VertexStyle → Black,
    VertexSize → 0.15, VertexLabels → "Name", ImagePadding → 10];
g = SetProperty[{g, 10}, VertexStyle → Red]
```



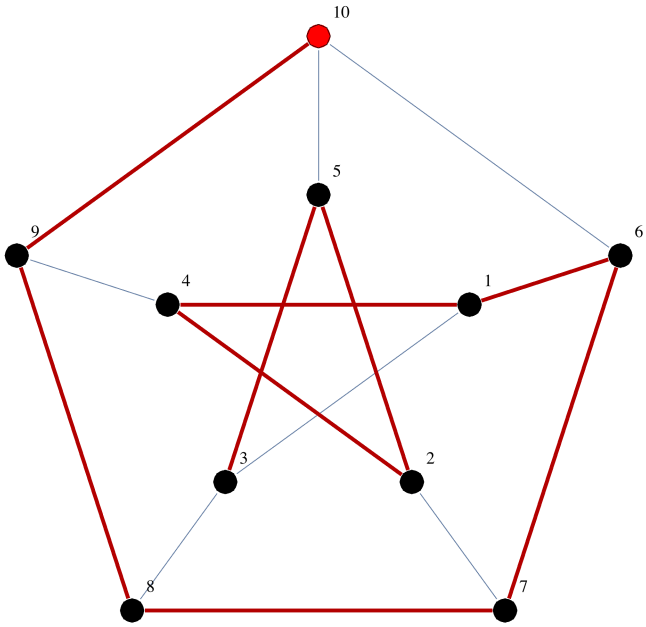

```
{parent, distance} = sDFS[g, 10]
```

```
{{6, 4, 5, 1, 2, 7, 8, 9, 10, 10}, {5, 7, 9, 6, 8, 4, 3, 2, 1, 0}}
```

```
Grid[{Prepend[Range[VertexCount[g]], "Distance"], Prepend[%[[2]], 10]},
 Dividers → {{2 → Red}, {2 → Red}}]
```

Distance	1	2	3	4	5	6	7	8	9	10
10	5	7	9	6	8	4	3	2	1	0

```
Transpose[{VertexList[g], parent}] /. {x_, y_} → UndirectedEdge[x, y];
HighlightGraph[g, %, GraphHighlightStyle → "Thick"]
```



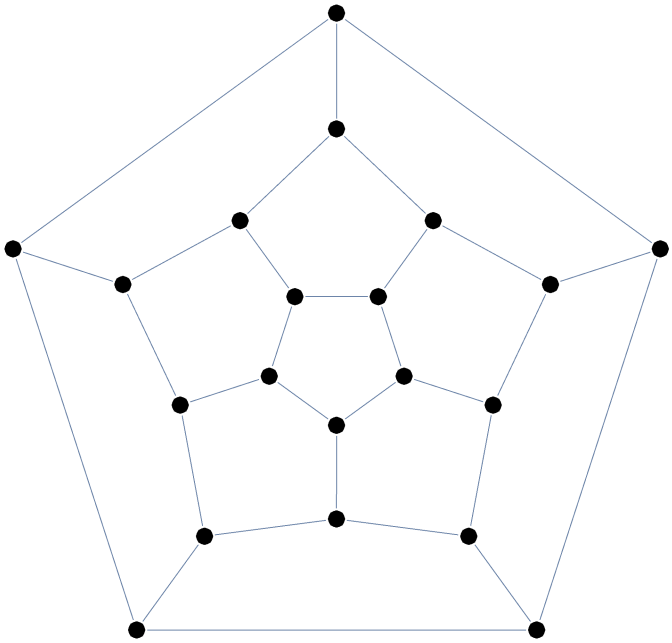
Note: `DepthFirsScan` is a built-in *Mathematica* function that implements the Depth-First Search Algorithm.

Hamiltonian Cyles

Definition 7.1. A **Hamiltonian path** in a graph G is a path which contains every vertex of G. A **Hamiltonian cycle** of a graph G is a cycle that contains every vertex of G. A graph G is called Hamiltonian if it has a Hamiltonian cycle.

In[12]:=
```
g = PolyhedronData["Dodecahedron", "SkeletonGraph"];
g = Fold[SetProperty[{#1, #2}, VertexStyle → Black] &, g, VertexList[g]];
g = Fold[SetProperty[{#1, #2}, VertexSize → Medium] &, g, VertexList[g]]
```

Out[14]=

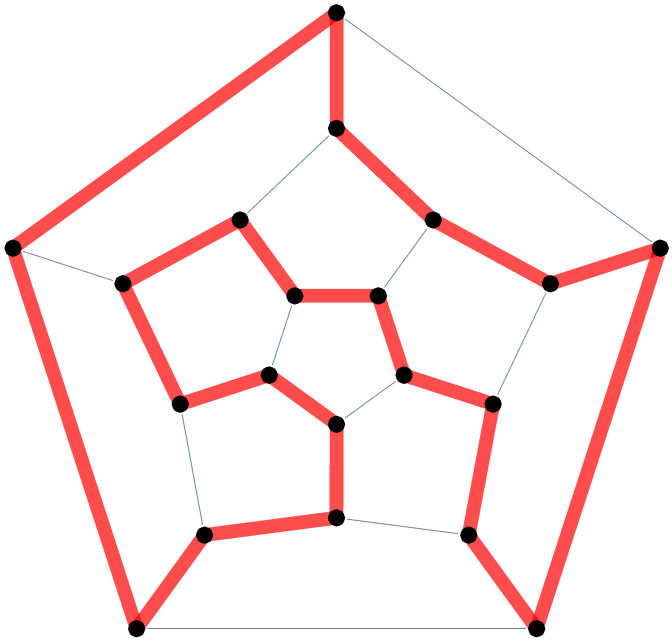


In[15]:=
```
hcycle = First[FindHamiltonianCycle[g]]
```

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

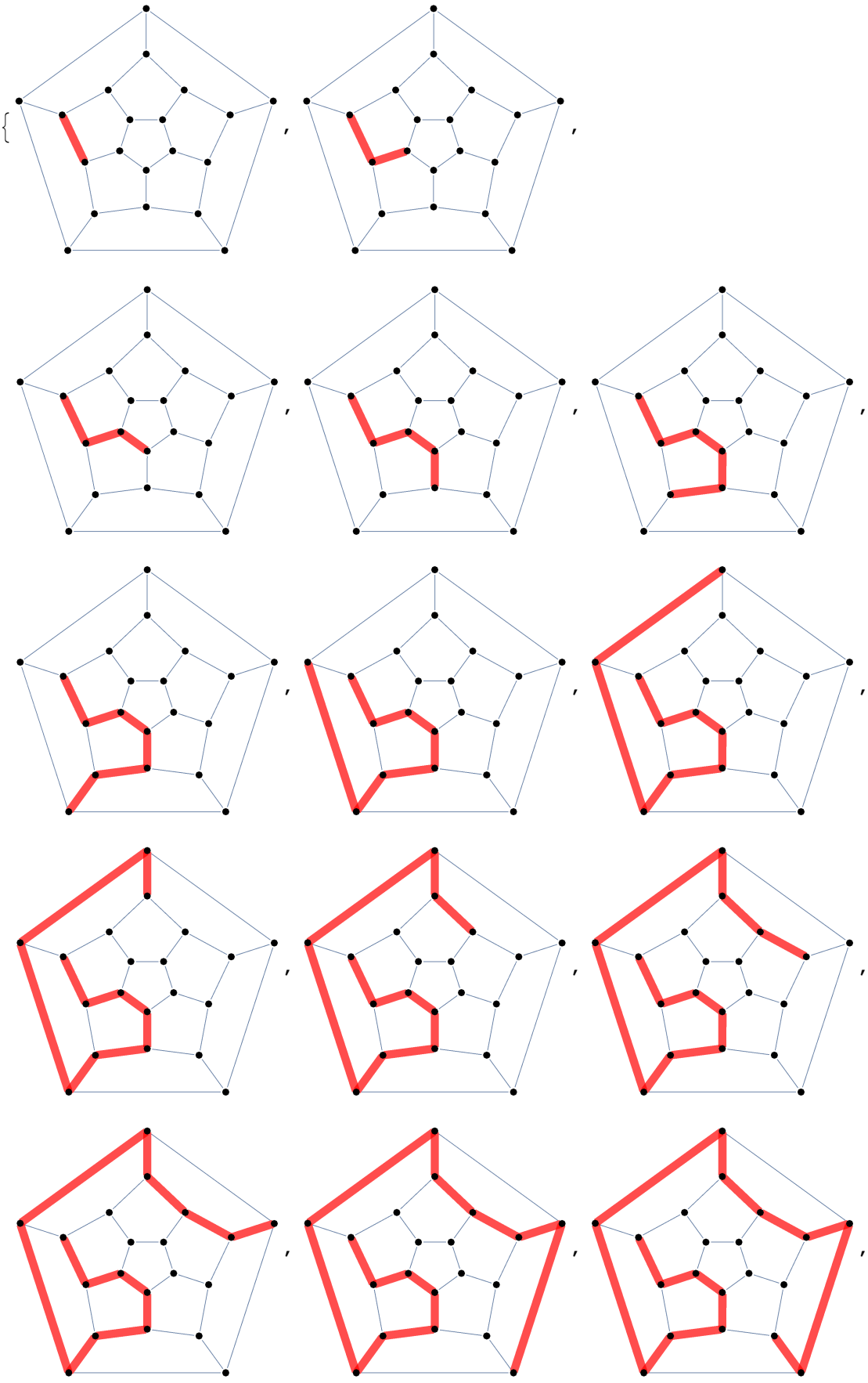
In[16]:=
```
HighlightGraph[g, Style[hcycle, {Thickness[0.02], Red}],
 VertexSize → Medium, VertexLabels → "Name"]
```

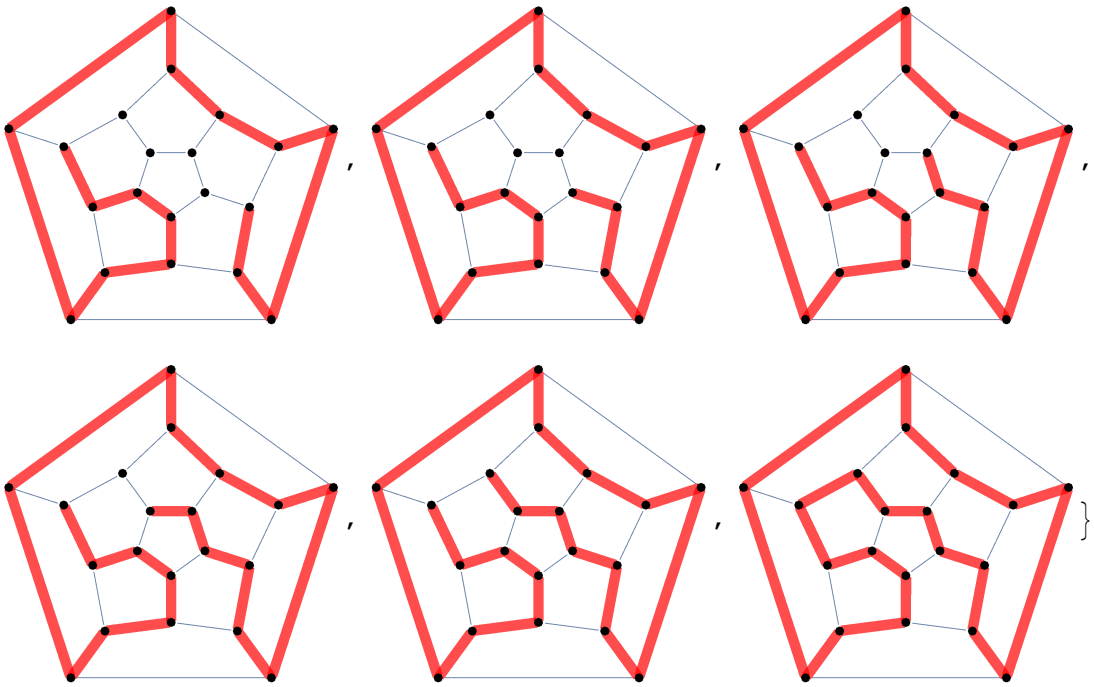
Out[16]=



In[17]:=
```
HighlightGraph[g, Style[#, {Thickness[0.03], Red}]] & /@
 Table[Take[hcycle, n], {n, 1, Length[hcycle]}]
```

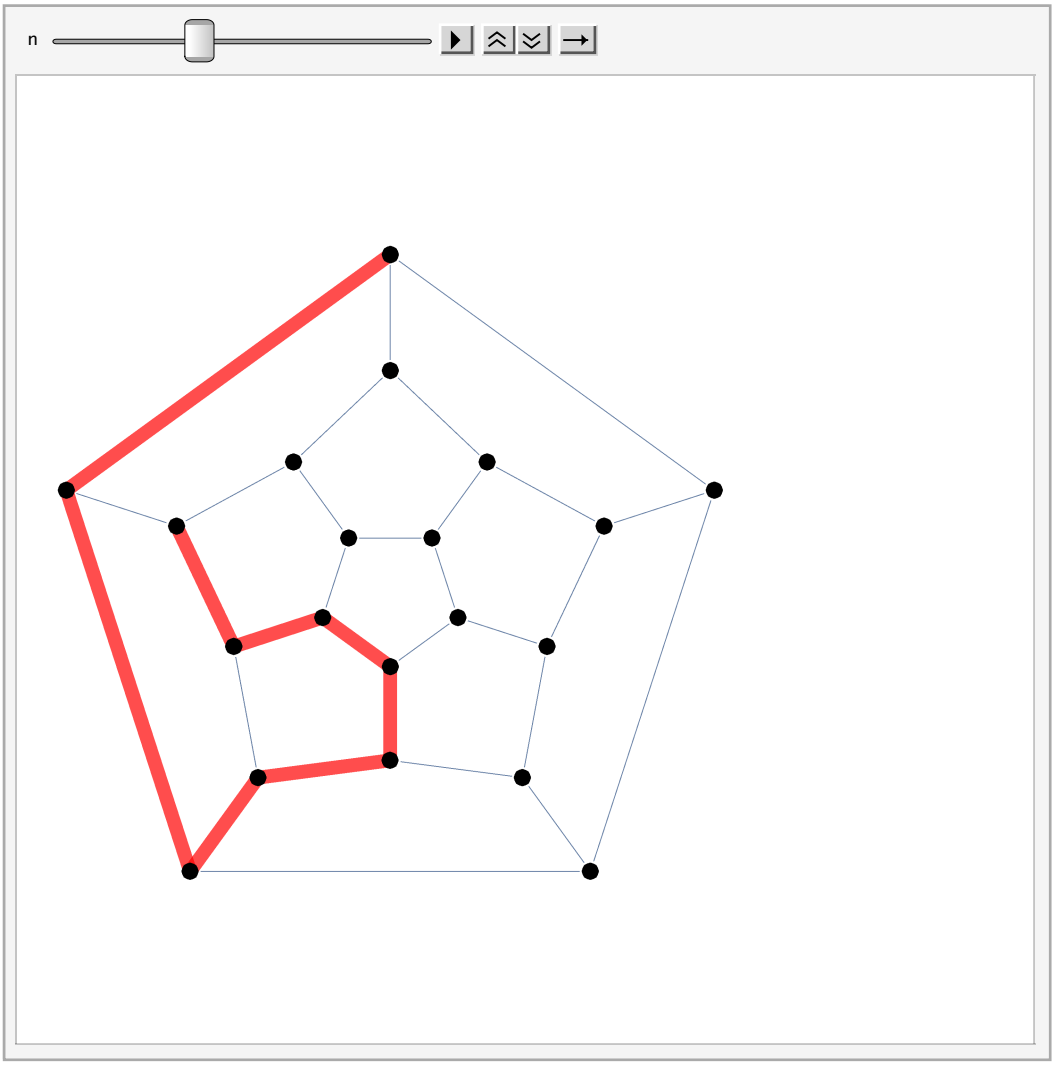
Out[17]=





In[18]:=
```
Animate[
 HighlightGraph[g, Style[Take[hcycle, n], {Thickness[0.02], Red}]],
 {n, 1, Length[hcycle], 1}, AnimationRunning → False]
```

Out[18]=

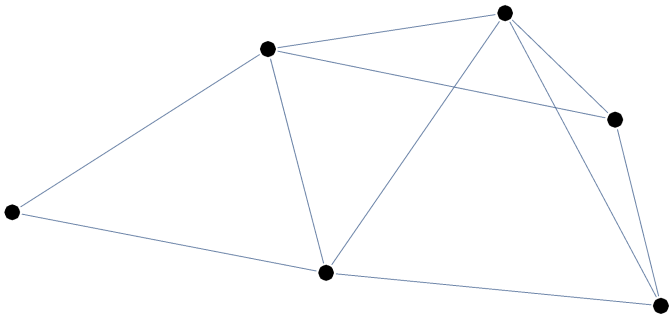


Euler Tours

Definition 8.1. A trail in G is called an **Euler trail** if it includes every edge of G. A graph is called **Eulerian** if it has a closed Euler trail.

In[1]:=
```
g1 = Graph[{1 ↔ 2, 1 ↔ 5, 1 ↔ 6, 2 ↔ 3, 2 ↔ 4, 2 ↔ 6, 3 ↔ 4, 4 ↔ 5, 4 ↔ 6, 5 ↔ 6},
  VertexStyle → Black, VertexSize → Small]
```

Out[1]=



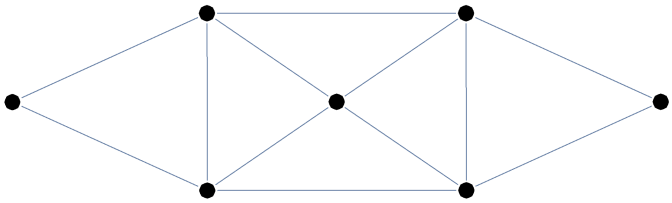
In[2]:=
```
Table[VertexDegree[g1, v], {v, VertexList[g1]}]
EulerianGraphQ[g1]
```

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

Out[3]= False

In[4]:=
```
g1 = Graph[{1 ↔ 2, 1 ↔ 5, 1 ↔ 6, 1 ↔ 7, 2 ↔ 3, 2 ↔ 4, 2 ↔ 6, 3 ↔ 4,
    4 ↔ 5, 4 ↔ 6, 5 ↔ 6, 5 ↔ 7}, VertexStyle → Black, VertexSize → Small]
```

Out[4]=



In[5]:=
```
Table[VertexDegree[g1, v], {v, VertexList[g1]}]
EulerianGraphQ[g1]
```

Out[5]= {4, 4, 4, 4, 2, 2, 4}

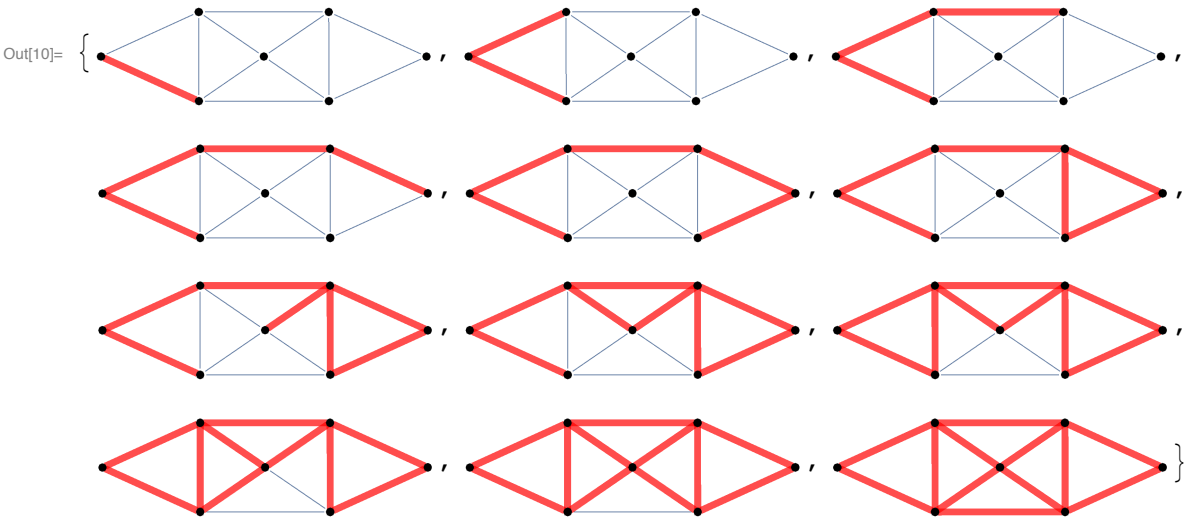
Out[6]= True

In[9]:=
```
eTour = First[FindEulerianCycle[g1]]
```

Out[9]= {1 ↔ 7, 7 ↔ 5, 5 ↔ 4, 4 ↔ 3, 3 ↔ 2, 2 ↔ 4, 4 ↔ 6, 6 ↔ 5, 5 ↔ 1, 1 ↔ 6, 6 ↔ 2, 2 ↔ 1}

In[10]:=
```
Table[HighlightGraph[g1, Style[Part[%, 1 ;; i], {Thickness[0.02], Red}]],
 {i, Length[eTour]}]
```

Out[10]=



In[11]:=
```
Animate[
 HighlightGraph[g1, Style[Take[eTour, n], {Thickness[0.02], Red}]],
 {n, 1, Length[eTour], 1}, AnimationRunning → False]
```

Out[11]=