10 Graphs

Introduction

In this chapter we consider ways in which *Mathematica* can help you explore and understand graphs. In particular, we describe how to do computations on graphs using *Mathematica* and how *Mathematica* can be used to visualize graphs.

Throughout the first half of this chapter, pseudographs are a recurring theme. Recall that pseudographs are graphs that may contain loops and may contain multiple edges between vertices. *Mathematica* includes numerous and powerful commands for representing, manipulating, and calculating with simple graphs, both undirected and directed. Each section in what follows will introduce you to these useful tools so that you can more easily explore the concepts described in the textbook. But many of these functions do not support graphs with multiple edges. So parts of this chapter are devoted to extending *Mathematica*'s existing functionality to pseudographs. This will give you tools that you can use to explore these kinds of graphs. More than that, seeing how to create the functions for pseudographs will help you to better understand how the functions work for simple graphs.

10.1 Graphs and Graph Models

Recall that a simple graph, as defined in Section 10.1 of the text, is a set V of vertices and a set E of unordered pairs of elements of V, called the edges of the graph, and where each edge connects two different vertices and no two edges connect the same pair of vertices. That is, the edges are undirected, there are no loops, and there are no multiple edges.

Mathematica represents a simple graph as a special raw object with head `Graph`. You should think about `Graph` not as a function, but as a way to describe and represent an object. To explain this distinction, consider a fraction, such as $\frac{5}{9}$. We think about this as a single number, and *Mathematica* treats it as a simple object, but if you delve down using `FullForm`, you see that it's a bit more complicated.

```
In[1]:= 5 / 9 // FullForm
Out[1]//FullForm=
        Rational[5, 9]
```

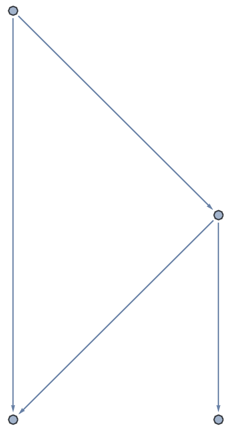
`FullForm` reveals that, for *Mathematica*, the fraction $\frac{5}{9}$ is actually represented by `Rational` applied to two integers. `Rational` is not a function, in the usual sense. Rather, it is a head that tells *Mathematica* what the contents mean. `Graph` is exactly the same, if a bit more complicated. It is a head that tells *Mathematica* that the contents represent a graph object. Also, just like a `Rational` is typically displayed in the usual fraction notation, a `Graph` object is displayed as a drawing of the graph.

The simplest way to specify a `Graph` object in *Mathematica* is by specifying the edges as a list of rules. You typically use positive integers or strings for the vertices (although other expressions can be used). An edge between the vertex represented by 1 and the vertex represented by "a" is given as the rule **`1 -> "a"`**.

As an example, the following defines a directed version of the graph shown in Exercise 3 in Section 10.1.

```
In[2]:= exercise3directed =
         Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"}]
```

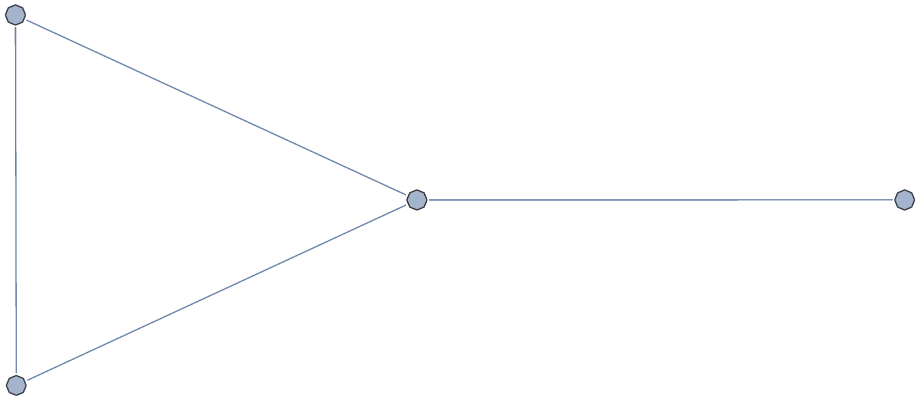
Out[2]=



To produce an undirected simple graph, you can set the option `DirectedEdges` to `False`.

```
In[3]:= exercise3 = Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
          DirectedEdges → False]
```

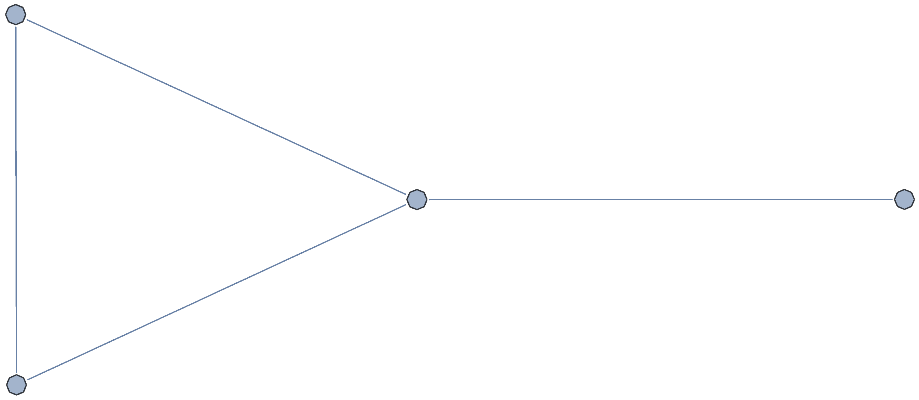
Out[3]=



You can also create (undirected) simple graphs using the symbol ⟷ in place of `Rule` (→). The undirected edge symbol is entered by typing ESCueESC.

```
In[4]:= Graph[{"a" •-• "b", "a" •-• "c", "b" •-• "c", "b" •-• "d"}]
```

Out[4]=

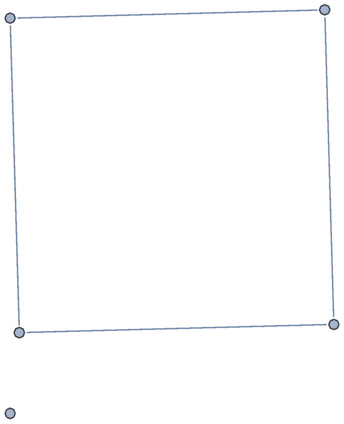


If you prefer, you can use the symbol •⟶, entered as `ESC`de`ESC` in place of `Rule` (→) for a directed graph. Note, however, that `Graph` does not allow mixed graphs, that is, graphs with both directed an undirected edges. In this manual, we will generally use `Rule` (→) to specify all edges together with the `DirectedEdges` option, when needed. There are, however, certain functions that require the use of •⟶ or •-•, and you should pay special attention to those situations.

If you wish, you may give an explicit list of vertices as an optional first argument to `Graph`. This is only required in the situation when a graph has a vertex not adjacent to any other, as in the example below.

```
In[5]:= Graph[{1, 2, 3, 4, 5},
          {1 → 2, 2 → 3, 3 → 4, 4 → 1}, DirectedEdges → False]
```

Out[5]=



The `VertexList` and `EdgeList` functions applied to a `Graph` object output the vertices and edges of the graph.

```
In[6]:= VertexList[exercise3]

Out[6]= {a, b, c, d}

In[7]:= EdgeList[exercise3]

Out[7]= {a •-• b, a •-• c, b •-• c, b •-• d}
```

Observe that the output of `EdgeList` is a list of undirected edges, using the symbol •-•, despite the fact that we defined the graph using rules. This is because the true effect of setting the `Direct-`

edEdges option to False is for *Mathematica* to transform the rules into undirected edges. Using FullForm, you can see that these are stored using the head UndirectedEdge.

```
In[8]:= EdgeList[exercise3] // FullForm
```

```
Out[8]//FullForm=
        List[UndirectedEdge["a", "b"], UndirectedEdge["a", "c"],
         UndirectedEdge["b", "c"], UndirectedEdge["b", "d"]]
```

Note that the same is true in the directed case, with the edges represented internally as DirectedEdge.

```
In[9]:= EdgeList[exercise3directed]
```

```
Out[9]= {a •→ b, a •→ c, b •→ c, b •→ d}
```

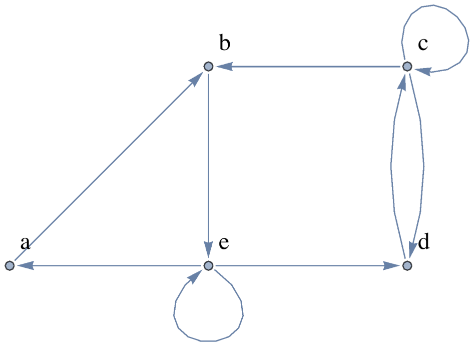
```
In[10]:= EdgeList[exercise3directed] // FullForm
```

```
Out[10]//FullForm=
        List[DirectedEdge["a", "b"], DirectedEdge["a", "c"],
         DirectedEdge["b", "c"], DirectedEdge["b", "d"]]
```

Note that Graph objects may contain loops, as illustrated below with a replica of Exercise 7 from Section 10.1. Loops are created simply by including an edge from a vertex to itself.

```
In[11]:= exercise7 = Graph[{"a", "b", "c", "d", "e"},
           {"a" → "b", "b" → "e", "c" → "b", "c" → "c",
            "c" → "d", "d" → "c", "e" → "a", "e" → "d", "e" → "e"},
           VertexLabels → "Name", ImagePadding → 5,
           VertexCoordinates → {{1, 1}, {2, 2}, {3, 2}, {3, 1}, {2, 1}}]
```

Out[11]=



Options Affecting the Visual Output of `Graph`

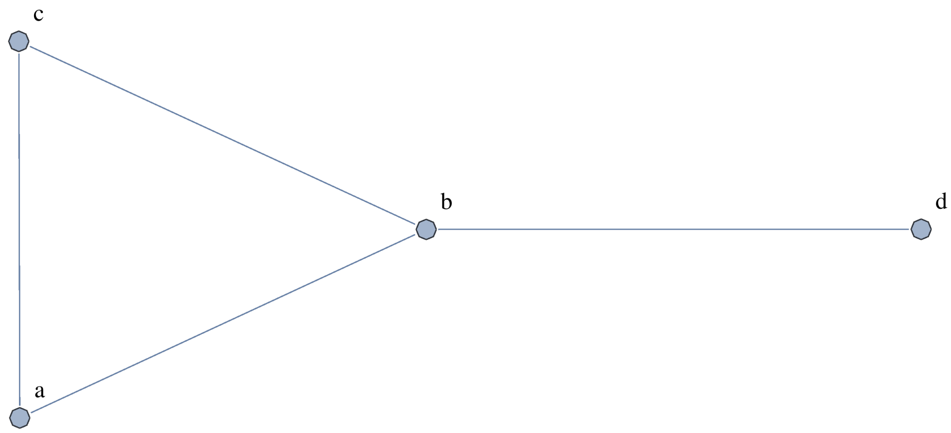
In this subsection, we will explain just a few of the options available when creating Graph objects to change their visual appearance. Readers interested in greater control over the visual display of Graph objects should refer to the help pages.

To have *Mathematica* display the names of vertices, you use the VertexLabels option. Probably the most common value for this option is **"Name"**, including the quotation marks, which causes each vertex to be labeled with its name. As an example, we repeat the definition of the graph from Exercise

3 used above, with this option invoked.

```
In[12]:= Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
          DirectedEdges → False, VertexLabels → "Name"]
```

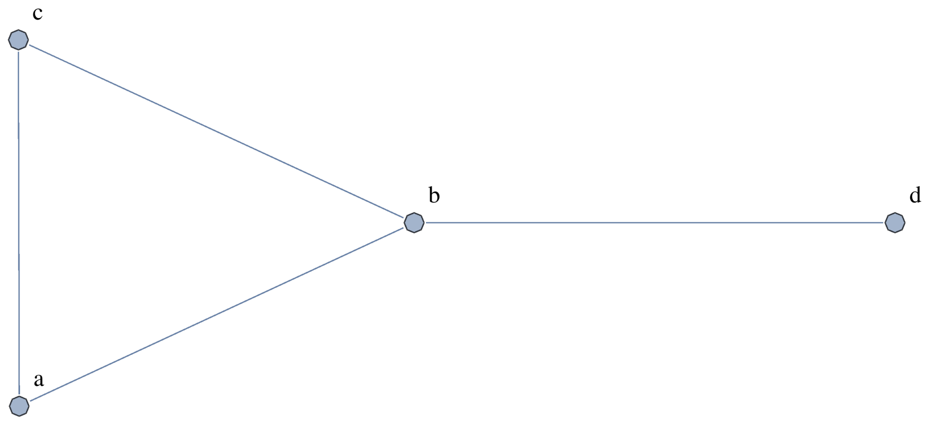
Out[12]=



Observe that *Mathematica* has cut off part of the string "c" which labels the top vertex. This is because it did not allocate room for the labels when it drew the graph. To correct for this, we apply the `ImagePadding` option, which is available for all graphics objects. The valid values for `ImagePadding` include: a single integer representing padding to be added to all sides of the image, measured in points; a list of the form `{{ left, right }, { bottom, top }}` indicating specific lengths for each side; `None`, for no padding, or `All`, which ensures there is enough padding for all the objects in the graphic. Below, we see that 5 points on all sides is sufficient in this example.

```
In[13]:= Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
          DirectedEdges → False, VertexLabels → "Name", ImagePadding → 5]
```

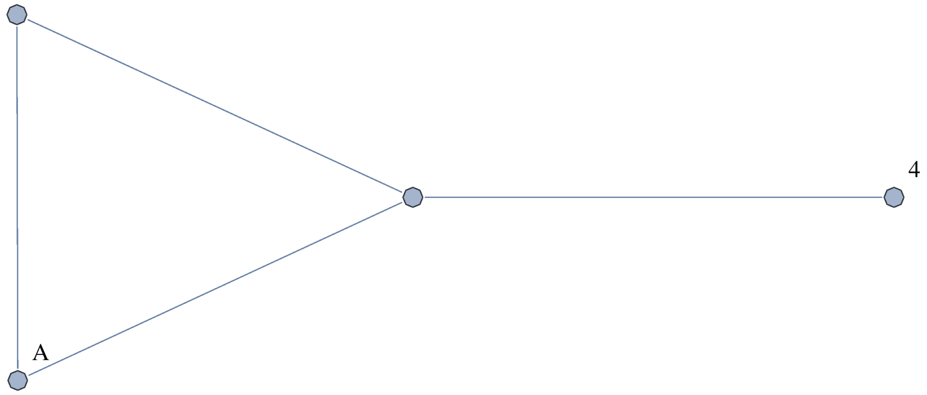
Out[13]=



Another use of the `VertexLabels` option is to specify labels for specific vertices. You do this by identifying the option with a list consisting of rules associating the name of the vertex with the desired label. The following illustrates this by labeling two of the vertices in the Exercise 3 graph.

```
In[14]:= Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
          DirectedEdges → False,
          VertexLabels → {"a" → "A", "d" → 4}, ImagePadding → 5]
```

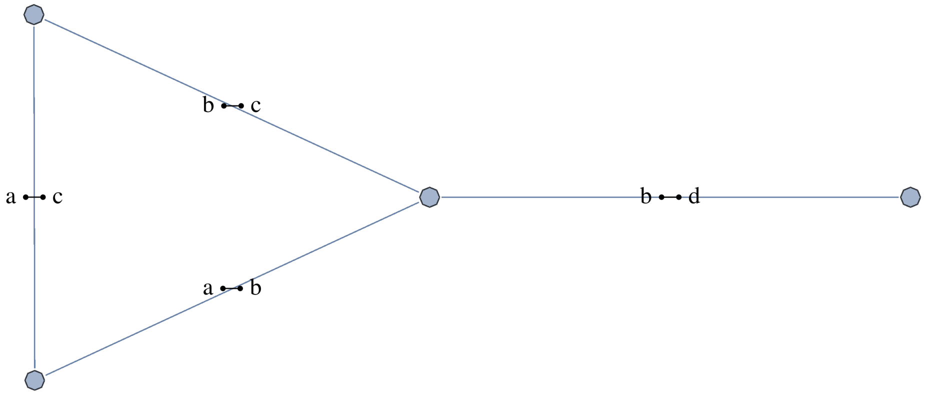
Out[14]=



Similarly, edges in a graph can be labeled using the `EdgeLabels` option. Once again, the value “Name” will cause all edges to be labeled.

```
In[15]:= Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
          DirectedEdges → False, EdgeLabels → "Name", ImagePadding → 5]
```

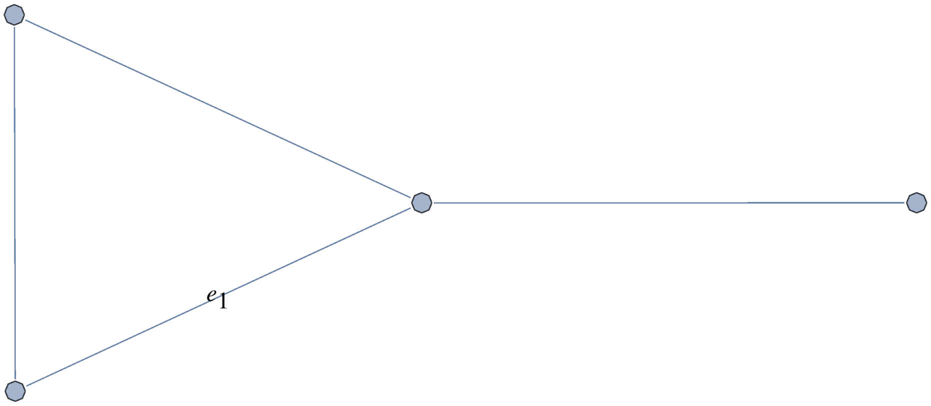
Out[15]=



As with vertices, you can also choose to label specific edges with arbitrary labels by giving a list of rules as the value to `EdgeLabels`. When doing this, the edges must be specified using either •–• (ESCueESC) or •⟶ (ESCdeESC).

In[16]:=
```
Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
 DirectedEdges → False, EdgeLabels → {"a" ⧟ "b" → e₁}]
```

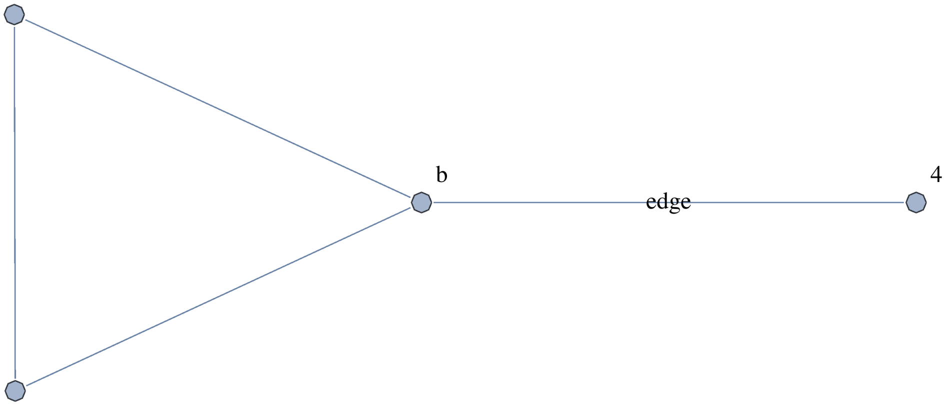
Out[16]=



As an alternative to `VertexLabels` and `EdgeLabels`, you can use the wrapper `Labeled` around an edge in the list of edges or, in conjuncion with the optional list of vertices, around a vertex. The first element in `Labeled` is the name of the vertex or the edge definition. The second element is the label to be used. Observe that, just as with the `EdgeLabels` option, edges within a `Labeled` wrapper must be given using ⧟ or ⟷.

In[17]:=
```
Graph[{"a", Labeled["b", "b"], "c", Labeled["d", 4]},
 {"a" → "b", "a" → "c", "b" → "c", Labeled["b" ⧟ "d", "edge"]},
 DirectedEdges → False]
```

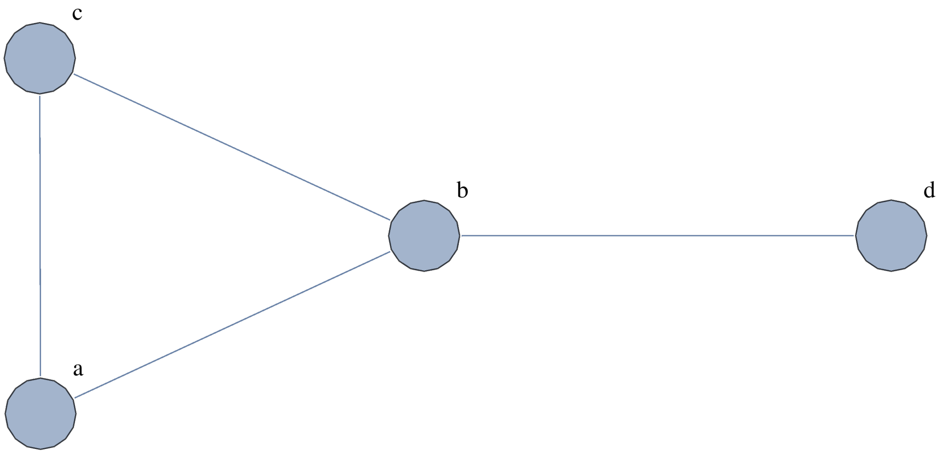
Out[17]=



You can change the size of vertices with the `VertexSize` option. Valid values include **`Tiny`**, **`Small`**, **`Medium`**, and **`Large`**, or a number between 0 and 1. A numerical value indicates that the size of the vertex should be that proportion of the distance between the closest two vertices.

```
In[18]:= Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
          DirectedEdges → False,
          VertexLabels → "Name", VertexSize → Medium]
```

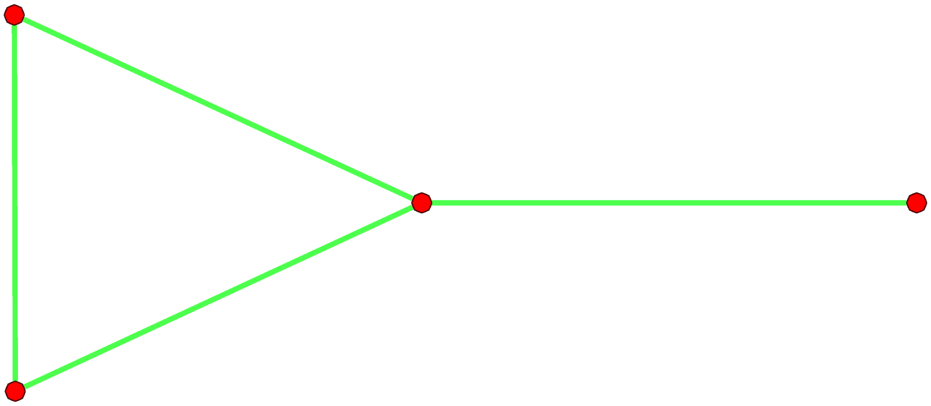
Out[18]=



You can further control the appearance, such as the color, of vertices and edges with the `VertexStyle` and `EdgeStyle` options. Multiple styles can be combined with the `Directive` wrapper. For example, the following illustrates how to create red vertices and thick green edges. Readers interested in exploring the various options available should consult the Graphics Directives guide.

```
In[19]:= Graph[{"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
          DirectedEdges → False, VertexStyle → Red,
          EdgeStyle → Directive[Green, Thick]]
```

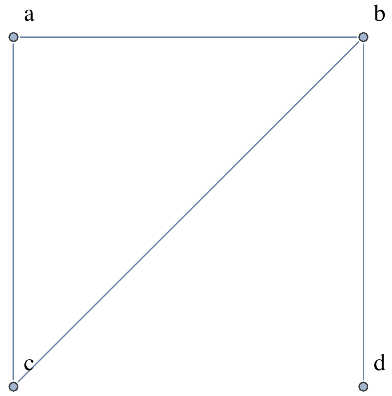
Out[19]=



The `VertexCoordinates` option is used to specify the locations of vertices. The value for the `VertexCoordinates` option is a list of pairs of the form $\{\{x_1, y_1\}, \{x_2, y_2\}, \ldots\}$, with the pairs specifying the coordinates of a vertex. The order of the locations must correspond to the order of the vertices in the graph. When an explicit list of vertices is given as the optional first element of `Graph`, that list specifies the order. Otherwise, the order is determined by when the vertex is first encountered in the list of edges, and is the same as the output of `VertexList`. Below, we use `VertexCoordinates` to redraw the Exercise 3 graph with the vertices in the same positions as in the image in the textbook.

```
In[20]:= Graph[{"a", "b", "c", "d"},
           {"a" → "b", "a" → "c", "b" → "c", "b" → "d"},
           DirectedEdges → False, VertexLabels → "Name",
           VertexCoordinates → {{1, 2}, {2, 2}, {1, 1}, {2, 1}},
           ImagePadding → 10]
```

Out[20]=



You can also exert control over the layout of the vertices of a graph, but without specifying each coordinate, by using the `GraphLayout` option to specify the algorithm that *Mathematica* uses to choose the vertex location. Common choices include: **`"SpringEmbedding"`**, which treats edges as springs and minimizes the mechanical energy of the system; **`"SpringElectricalEmbedding"`**, which treats edges as springs and vertices as electrical charges and minimizes mechanical and electrical energy; **`"HighDimensionalEmbedding"`**, which is like the spring-electrical method but computes in high dimensions and then projects down to two; **`"CircularEmbedding"`**, which places the vertices on a circle; and **`"LayeredDrawing"`**, which places vertices in layers and attempts to reduce the number of edges between non-adjacent layers.

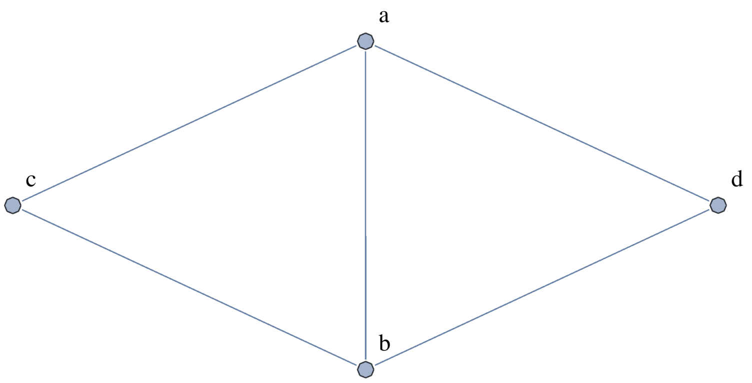
Modifying Graphs

You can modify existing graphs by adding and deleting edges and vertices using the functions `VertexAdd`, `VertexDelete`, `EdgeAdd`, and `EdgeDelete`. Each of these functions requires a graph as the first argument. The second argument is usually either a vertex, an edge, or a list of vertices or edges. The deletion functions can, in place of a vertex, edge, or list, accept a pattern as the second argument in order to delete all of the vertices or edges that match the pattern.

First, we create an example graph to illustrate the functions with.

```
In[21]:= modifyExample =
          Graph[{"a" → "b", "a" → "c", "a" → "d", "b" → "c", "b" → "d"},
           DirectedEdges → False,
           VertexLabels → "Name", ImagePadding → 10]
```

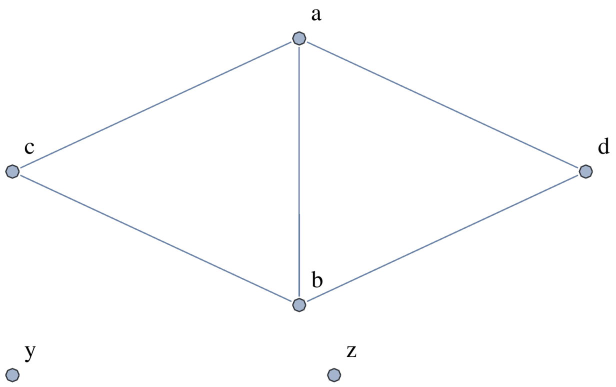
Out[21]=



Now we use `VertexAdd` to add two new vertices to the graph.

```
In[22]:= modifyExampleB = VertexAdd[modifyExample, {"y", "z"}]
```

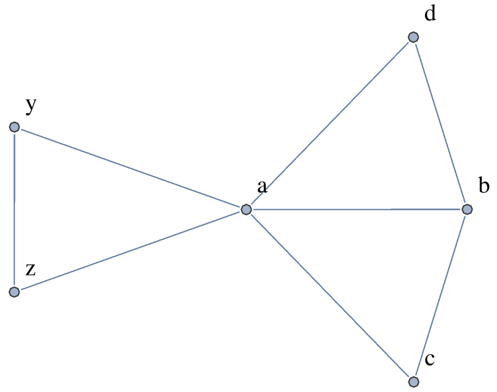
Out[22]=



Now we add edges to connect the new vertices with the rest of the graph. Note that *Mathematica* will interpret rules as undirected edges since the original graph is not directed.

```
In[23]:= modifyExampleB =
          EdgeAdd[modifyExampleB, {"a" → "z", "a" → "y", "y" → "z"}]
```

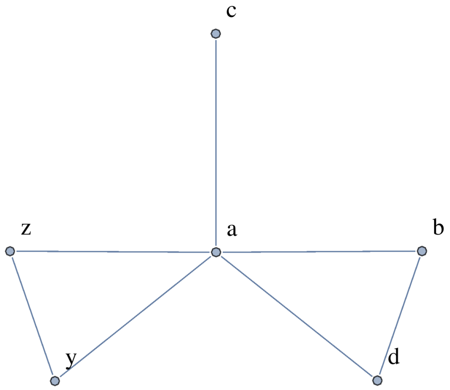
Out[23]=



Now we delete one of the old edges. Note that to delete an edge from an undirected graph, you must use •–• (ESCueESC). Likewise, deleting an edge from a directed graph must use •→ (ESCdeESC).

```
In[24]:= modifyExampleB = EdgeDelete[modifyExampleB, "b" •–• "c"]
```

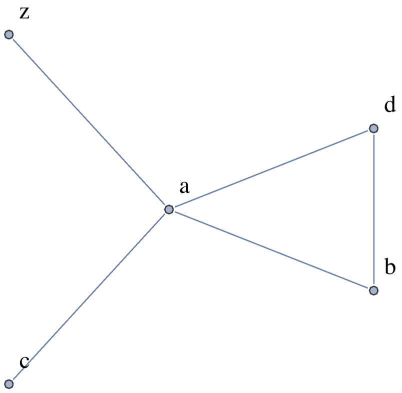
Out[24]=



Finally, note that deleting a vertex also deletes all the edges incident with that vertex.

```
In[25]:= modifyExampleB = VertexDelete[modifyExampleB, "y"]
```

Out[25]=



```
In[26]:= VertexList[modifyExampleB]

Out[26]= {a, b, c, d, z}

In[27]:= EdgeList[modifyExampleB]

Out[27]= {a •-• b, a •-• c, a •-• d, b •-• d, a •-• z}
```

Multiple edges and `GraphPlot`

We saw that loops are allowed, but `Graph` objects may not contain multiple edges. The following is the list of edges for Exercise 4 from Section 10.1.

```
In[28]:= exercise4edges = {"a" → "b", "a" → "b",
            "a" → "c", "b" → "d", "b" → "d", "b" → "d", "c" → "d"};
```

This cannot be used to form a `Graph`.

```
In[29]:= Graph[exercise4edges]

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

Out[29]= Graph[{a → b, a → b, a → c, b → d, b → d, b → d, c → d}]
```

As mentioned above, *Mathematica* includes many functions that can be applied to `Graph` objects in order to check graph properties and do other computations with them. In order to work with multigraphs, that is, graphs with multiple edges, we will build our own functions to do computations. Fortunately, we do not need to develop a new function for drawing images of multigraphs.

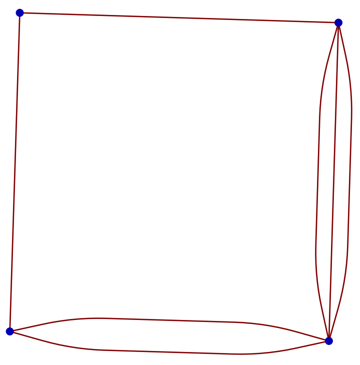
The `GraphPlot` function, having been introduced in *Mathematica* version 6, is older than and partially superceded by the `Graph` object. However, it has some benefits, in particular the ability to draw multigraphs.

The basic input for `GraphPlot` is a list of rules representing the edges of the graph. In this regard, `GraphPlot` and `Graph` are similar. However, `GraphPlot` does not allow the use of the symbols •-• or •→, nor does it allow a list of vertices as a first argument.

You can draw a plot of Exercise 4 by applying `GraphPlot` to **`exercise4edges`**.

```
In[30]:= GraphPlot[exercise4edges]
```

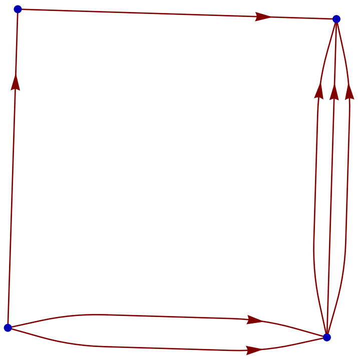
Out[30]=



Right away, you can see another important difference between `GraphPlot` and `Graph`. Specifically, where Graph's default behavior is to interpret rules as directed edges, GraphPlot assumes that rules are not directed. To draw a directed graph, you must use the DirectedEdges option.

```
In[31]:= GraphPlot[exercise4edges, DirectedEdges → True]
```

Out[31]=

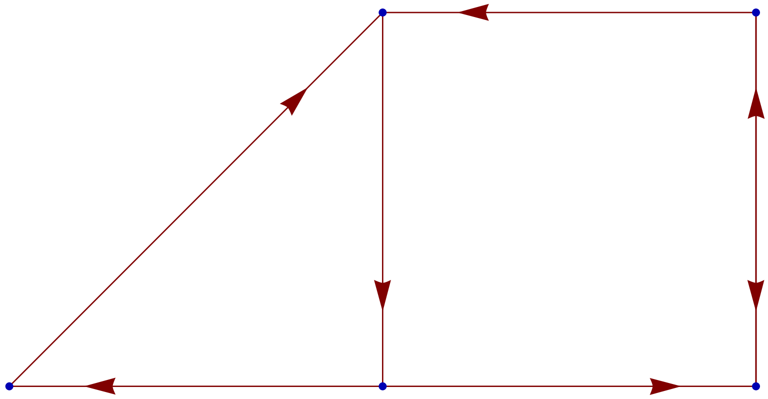


The most important difference between GraphPlot and Graph is the fact that where Graph produces a graph object, which can be computed with and manipulated, GraphPlot only produces a graphics object, just like Plot or Plot3D. For this reason, we can not assign a symbol to the result of GraphPlot and use it to do computations on the graph. Instead, when dealing with multigraphs, we will treat the list of rules representing the graph's edges, such as **exercise4edges**, as the representation of the graph.

One final comment about the relationship between GraphPlot and Graph: GraphPlot can often be given a Graph object as its argument in order to create a plot of the graph. For example, we can apply GraphPlot to **exercise7.**

```
In[32]:= GraphPlot[exercise7]
```

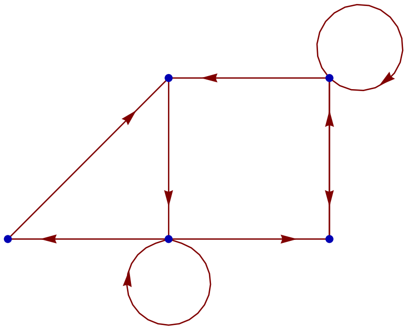
Out[32]=



This preserves some of the options from the Graph object, such as the vertex positions, but discards options that GraphPlot does not implement or are implemented differently, such as the vertex labels. You may also observe that the GraphPlot version has removed the loops. These can be displayed by explicitly setting the SelfLoopStyle option to All.

```
In[33]:= GraphPlot[exercise7, SelfLoopStyle → All]
```

Out[33]=



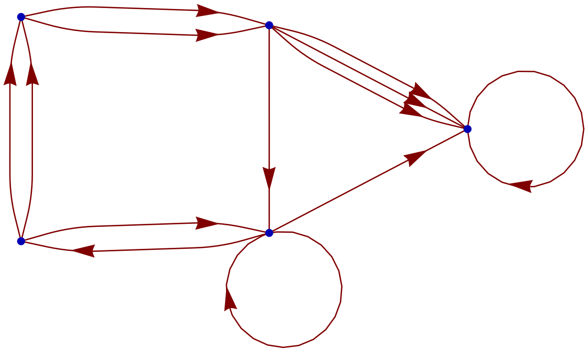
We now focus on some of the options you can use with `GraphPlot` to control the plot’s appearance. Note that many of these are different from and incompatible with the corresponding option for `Graph`. We will illustrate with Exercise 8 from Section 10.1, whose edge list we produce below.

```
In[34]:= exercise8edges = {"a" → "b", "a" → "b", "a" → "e",
           "b" → "c", "b" → "c", "c" → "d", "c" → "d", "c" → "d",
           "c" → "e", "d" → "d", "e" → "a", "e" → "d", "e" → "e"}

Out[34]= {a → b, a → b, a → e, b → c, b → c, c → d,
          c → d, c → d, c → e, d → d, e → a, e → d, e → e}
```

We have already mentioned the `DirectedEdges` option. By default, `GraphPlot` draws edges as undirected. We set `DirectedEdges` to `True` to draw the edges as arrows.

```
In[35]:= GraphPlot[exercise8edges, DirectedEdges → True]
```

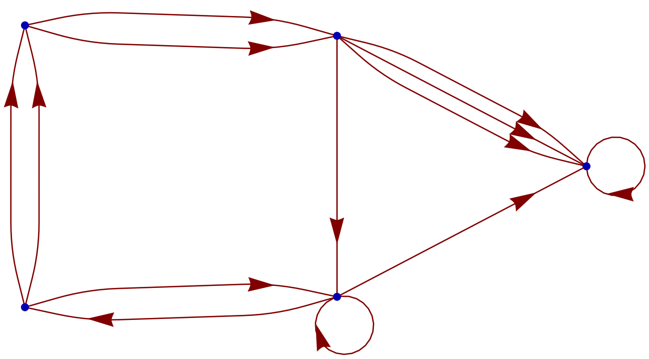
Out[35]=



We have also already mentioned the `SelfLoopStyle` option. The value `All` ensures that all self-loops are drawn, while `None` will cause the loops to not be drawn. You can also specify a value between 0 and 1, which will scale the loops relative to the average length of edges in the graph.

```
In[36]:= GraphPlot[exercise8edges,
          DirectedEdges → True, SelfLoopStyle → 0.2]
```

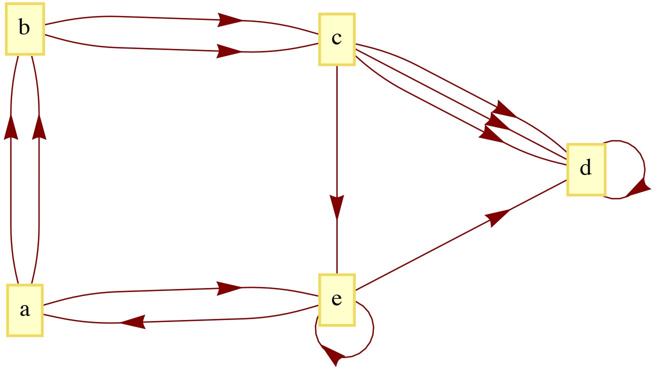
Out[36]=



To display the names of vertices, you use the `VertexLabeling` option. The default behavior, provided the number of vertices is not too large, is for the names of vertices to appear in a tooltip when you move the mouse pointer over the vertex. This option can be explicitly given with value `Tooltip`. By setting the `VertexLabeling` option to `True`, the names will be displayed on the graph itself.

```
In[37]:= GraphPlot[exercise8edges, DirectedEdges → True,
          SelfLoopStyle → 0.2, VertexLabeling → True]
```

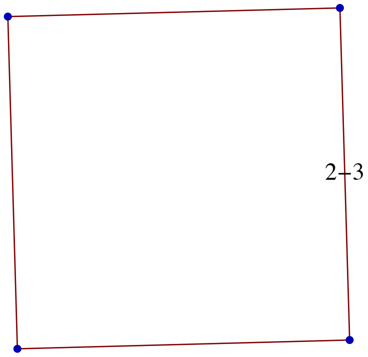
Out[37]=



To display labels for edges, you must explicitly provide the labels. This is done by replacing the rule defining an edge by a list consisting of the rule and an expression for the label. We illustrate with the following simple example.

```
In[38]:= GraphPlot[{1 → 2, {2 → 3, "2-3"}, 3 → 4, 4 → 1}]
```

Out[38]=

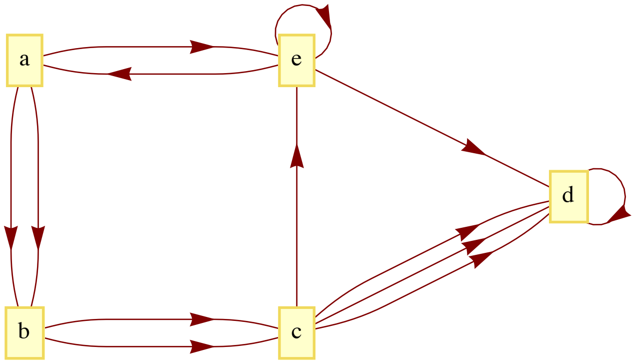


The option `EdgeLabeling` can be set to `Automatic` to instead display the label only as a tooltip, or to `False` to suppress display of the labels.

To explicitly control the placement of vertices, you use the `VertexCoordinateRules` option. The value of this option is a list of rules identifying the name of a vertex with a pair representing x and y coordinates. You can use the symbol `Automatic` in place of the coordinate pair to indicate that *Mathematica* should determine the location for that vertex automatically. This is the default if vertices are omitted from the rules. You can also use `Automatic` in place of either the x or y value within a pair if you wish to specify one value but leave the other to *Mathematica* to determine. We use `VertexCoordinateRules` to rearrange the vertices in the Exercise 8 example to match the locations in the textbook.

```
In[39]:= GraphPlot[exercise8edges, DirectedEdges → True,
          SelfLoopStyle → 0.2, VertexLabeling → True,
          VertexCoordinateRules → {"a" → {0, 1}, "b" → {0, 0},
            "c" → {1, 0}, "d" → {2, 0.5}, "e" → {1, 1}}]
```

Out[39]=



Finally, the `Method` option can be used to specify the algorithm used to lay out the vertices. The values are very similar to `GraphLayout` including **“SpringEmbedding”**, **“SpringElectricalEmbedding”**, **“HighDimensionalEmbedding”**, and **“CircularEmbedding”**.

10.2 Graph Terminology and Special Types of Graphs

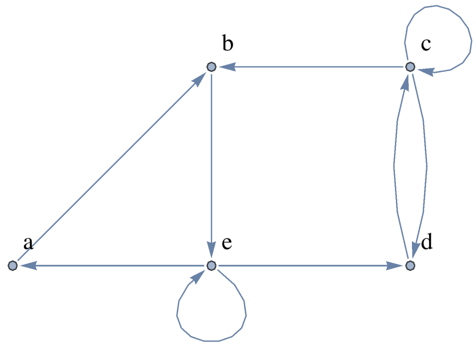
In this section we will see how to use *Mathematica* to perform computations related to some of the basic terminology of graphs, such as calculating degree. We will also look at some of the special families of graphs that *Mathematica* has built-in support for. And we discuss subgraphs and unions of graphs in *Mathematica*.

Degree

For a `Graph` object, *Mathematica* includes the function `VertexDegree` for determining the degree of a vertex. Given a `Graph` object and one of the graph's vertices, the function returns the number of edges incident to that vertex. For example, we can check the degrees of vertices *a* and *e* of **exercise7** from the previous section.

```
In[40]:= exercise7
```

Out[40]=



```
In[41]:= VertexDegree[exercise7, "a"]

Out[41]= 2

In[42]:= VertexDegree[exercise7, "e"]

Out[42]= 5
```

Observe that the loop at the vertex *e* counts as 2 towards the degree of that vertex. Also note that with this directed graph, `VertexDegree` calculates the number of edges incident to the given vertex without regard for their direction. *Mathematica* provides `VertexInDegree` and `VertexOutDegree` functions for calculating the directed values. As an example, consider vertex *d* from above.

```
In[43]:= VertexInDegree[exercise7, "d"]

Out[43]= 2

In[44]:= VertexOutDegree[exercise7, "d"]

Out[44]= 1
```

All three of these functions can be used without a second argument. If they are passed only the graph as the sole argument, they will return a list of the degrees of the vertices. Note that the output is in the same order as the output from `VertexList`.

```
In[45]:= VertexList[exercise7]

Out[45]= {a, b, c, d, e}

In[46]:= VertexDegree[exercise7]

Out[46]= {2, 3, 5, 3, 5}

In[47]:= VertexInDegree[exercise7]

Out[47]= {1, 2, 2, 2, 2}

In[48]:= VertexOutDegree[exercise7]

Out[48]= {1, 1, 3, 1, 3}
```

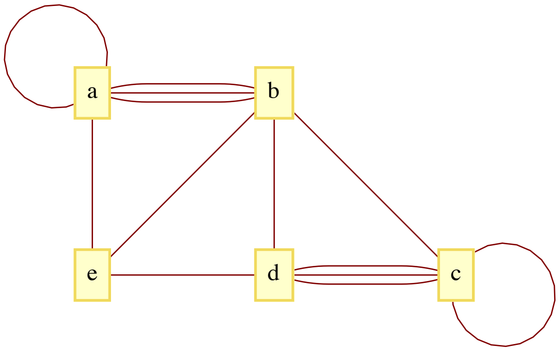
Degree in Pseudographs

Mathematica's built-in functions for degree cannot take into account multiple edges. We will write a function to rectify this, at least for the undirected degree. The in-degree and out-degree functions for directed graphs are left to the reader.

We reproduce Exercise 2 from Section 10.2 to use as an example. Recall that the presence of multiple edges means that we cannot create a `Graph` object. Instead, we model the graph as a list of edges, with multiple edges repeated. We will use `GraphPlot` to display an image of the graph, but our function for computing degree will take the list of edges as the input.

```
In[49]:= exercise2edges = {"a" → "a", "a" → "b", "a" → "b",
            "a" → "b", "a" → "e", "b" → "c", "b" → "d", "b" → "e",
            "c" → "c", "c" → "d", "c" → "d", "c" → "d", "d" → "e"};

In[50]:= GraphPlot[exercise2edges, VertexLabeling → True,
          VertexCoordinateRules → {"a" → {0, 1},
            "b" → {1, 1}, "c" → {2, 0}, "d" → {1, 0}, "e" → {0, 0}}]
```

Out[50]=



Note that we use the `VertexLabeling` option to display the names of the vertices and the `VertexCoordinateRules` option to ensure that the position of vertices corresponds to the image in the textbook. Using the `DirectedEdges` option is unnecessary since `GraphPlot` defaults to undirected edges.

To calculate the degree of a specified vertex, given the list of edges, we must count the number of edges in which the vertex is an endpoint. Keep in mind that a loop contributes twice to the degree of the vertex. This means that the degree of a vertex is the number of times that the vertex name appears in the list of edges as either endpoint.

We will use the built-in function `Count` to count the number of times the vertex appears. `Count` requires two arguments: a list (or other expression) within which to count and a pattern being sought. It returns the number of times the pattern matches an element of the list. A third argument allows you to specify the level at which to search. By default, `Count` will only count the elements of the list that match the pattern. In this case, we want to count elements of `Rule`s that are members of the list. That is, we want `Count` to look at level 2. To specify this, we give `Count` the third argument **`{2}`**, indicating that it should only count expressions at the second level that match the pattern.

So to find the degree of *a*, we apply `Count` to the edge list, the name of the vertex **`"a"`**, and **`{2}`**.

```
In[51]:= Count[exercise2edges, "a", {2}]
```

```
Out[51]= 6
```

We use this approach to create a function as shown below.

```
In[52]:= undirectedDegree[edgeList : {___Rule}, vertex_] :=
          Count[edgeList, vertex, {2}]
```

We check the function by applying it to the vertex *d*.

```
In[53]:= undirectedDegree[exercise2edges, "d"]
```

```
Out[53]= 5
```

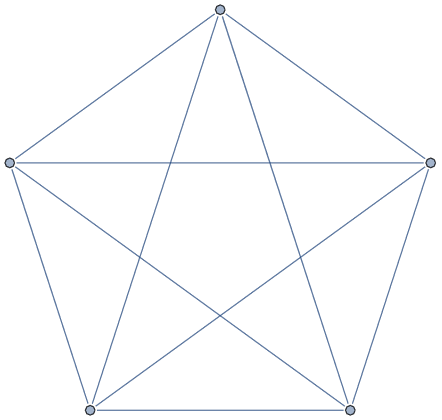
Some Special Simple Graphs

The textbook discusses several families of graphs, including complete graphs, cycles, and wheels. *Mathematica* provides functions for easily creating these and other special graphs.

We begin with complete graphs. Recall that a complete graph is a simple, undirected graph on a given number of vertices that has all possible edges between those vertices. The complete graph on *n* vertices is denoted K_n. The complete graph on *n* vertices can be obtained with the function `CompleteGraph` applied to *n*. For example, we can generate and display K_5, the complete graph on 5 vertices.

```
In[54]:= CompleteGraph[5]
```

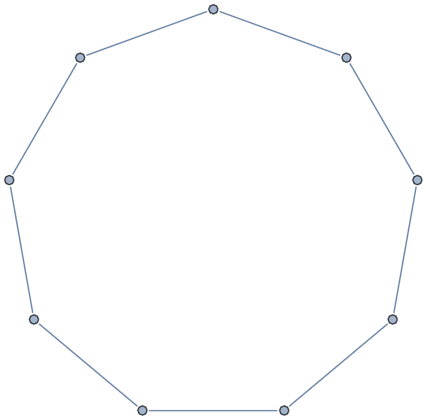
Out[54]=



Similarly, the cycle C_n is obtained with the function `CycleGraph`.

In[55]:=
```
CycleGraph[9]
```

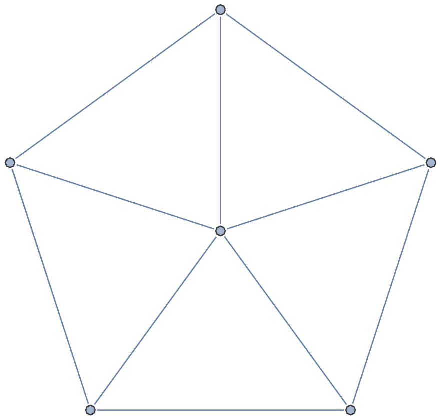
Out[55]=



A wheel W_n is obtained from the cycle graph C_n by adding one additional vertex adjacent to all n of the original vertices. In *Mathematica*, wheel graphs are obtained by `WheelGraph` applied to the value $n+1$, the total number of vertices in the wheel, not just the outside ring.

In[56]:=
```
WheelGraph[6]
```

Out[56]=

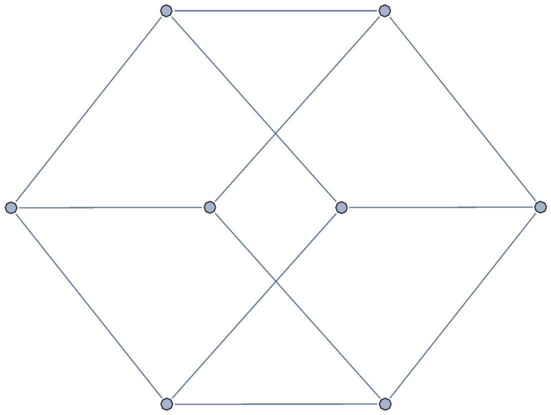


Hypercubes

To construct the n-cube Q_n, we use the `HypercubeGraph` function applied to the dimension n. Recall the definition of the hypercube graph given in Example 8 of Section 10.2. There are 2^n vertices labeled with the binary representations of the numbers 0 through $2^n - 1$. Two vertices are adjacent if their binary representations differ in only one digit. Here is the presentation of the three dimensional cube.

```
In[57]:= HypercubeGraph[3]
```

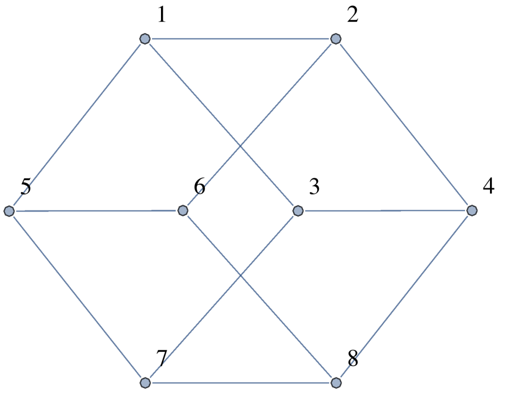
Out[57]=



By default, the vertices are not labeled. To have *Mathematica* label the vertices, we can use the `VertexLabels` option. In fact, `HypercubeGraph` accepts all the options that `Graph` does.

```
In[58]:= HypercubeGraph[3, VertexLabels → "Name", ImagePadding → 10]
```

Out[58]=



One more modification will allow us to see the connection between this image and the definition. Instead of using **"Name"** as the argument to `VertexLabels`, we can specify the labels explicitly by setting the option to a list of rules identifying the integers with the binary expression.

The definition of Q_n tells us that the vertices should be considered to be the binary representations of the integers from 0 to 7. So we will apply labels by subtracting 1 from the integer vertex names and using the `IntegerString` function to obtain the binary representation. The `IntegerString` function requires two arguments, an integer and a base, and produces a string representing the integer in that base. A third optional argument allows you to specify a minimal length for the string.

The following `Table` produces the list of rules identifying the vertex names with the appropriate label.

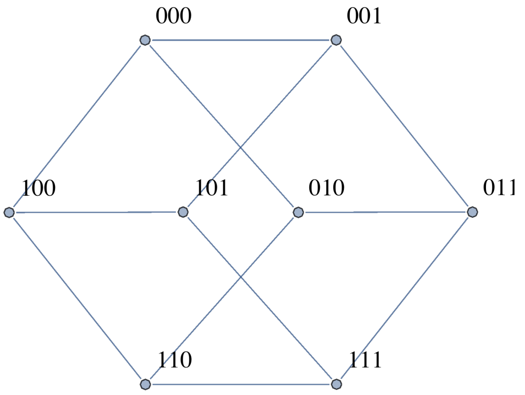
```
In[59]:= Table[v → IntegerString[v - 1, 2, 3], {v, 8}]
```

```
Out[59]= {1 → 000, 2 → 001, 3 → 010,
          4 → 011, 5 → 100, 6 → 101, 7 → 110, 8 → 111}
```

Using this as the value for `VertexLabels` produces a graph similar to the one shown in Figure 6 of Section 10.2 of the textbook.

```
In[60]:= HypercubeGraph[3,
          VertexLabels → Table[v → IntegerString[v - 1, 2, 3], {v, 8}],
          ImagePadding → 10]
```

Out[60]=



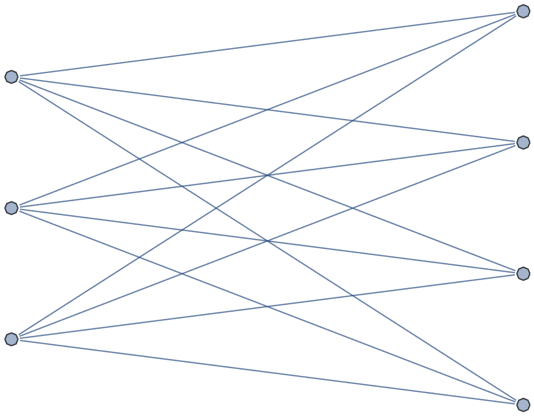
Bipartite Graphs

Another important class of graphs is the bipartite graphs. A bipartite graph is one whose vertex set can be partitioned into two disjoint sets such that every edge has one vertex in each of the partitioning sets. In other words, no two vertices in the same partitioning set are adjacent. We write $V = (A, B)$ to indicate that the vertex set V is partitioned into the sets A and B.

The complete bipartite graph $K_{m,n}$ is a bipartite graph with bipartition $V = (A, B)$ such that there are m vertices in A and n in B and such that there is an edge for every pair of vertices $a \in A$ and $b \in B$. The `CompleteGraph` function can be used to create complete bipartite graphs. The argument is the list consisting of the pair of m and n.

```
In[61]:= CompleteGraph[{3, 4}]
```

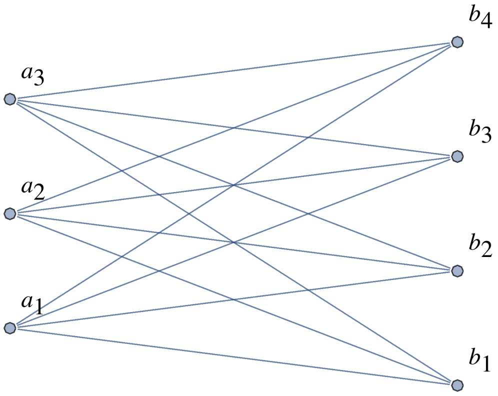
Out[61]=



Notice that *Mathematica* draws the complete bipartite graph with the two partitioning sets along the left and right to make the partition visually clear. As with the other functions in this section, the usual options for `Graph` apply. We illustrate how to label the vertices in a meaningful way. To enter a subscript in *Mathematica*, press CTRL and -.

```
In[62]:= CompleteGraph[{3, 4}, VertexLabels → Union[Table[i → a_i, {i, 3}],
           Table[i → b_{i-3}, {i, 4, 7}]], ImagePadding → 10]
```

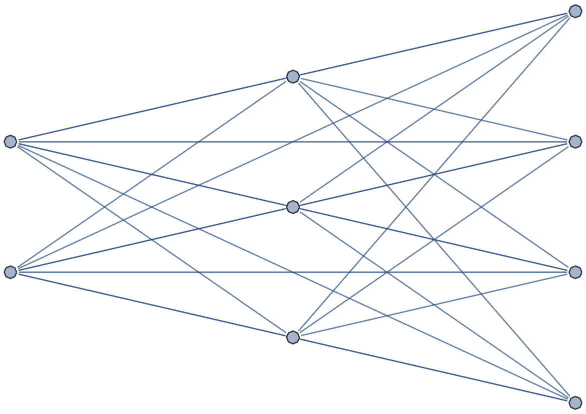
Out[62]=



Mathematica can also produce complete multipartite graphs. A *k*-partite graph is a graph in which the vertices can be partitioned into *k* disjoint sets so that no two vertices in any one of the partitioning sets are adjacent.

```
In[63]:= CompleteGraph[{2, 3, 4}]
```

Out[63]=



Mathematica has a function for determining whether a given graph is bipartite: `BipartiteGraphQ`. This function accepts a graph as its sole argument and returns `True` if the graph is bipartite.

```
In[64]:= BipartiteGraphQ[HypercubeGraph[3]]

Out[64]= True
```

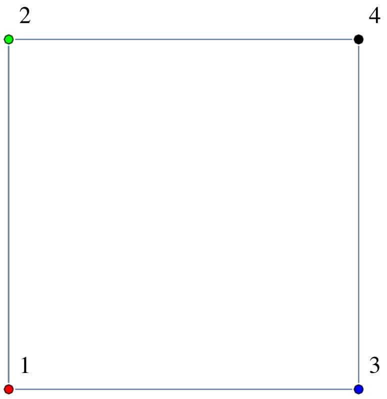
Bipartite Pseudographs

It is worthwhile, however, to recreate a version of **`BipartiteGraphQ`** from scratch in order to better understand the algorithm that determines whether the graph is bipartite and finds a bipartition. Our function will apply to edge lists, so as to be applicable to pseudographs. Also, instead of just returning true, our function will, if the graph is bipartite, display the graph with the vertices colored red and green to represent the partitioning. Of course, if the graph is not bipartite, the function will return false.

For `Graph` objects, the color of vertices can be changed by setting the `VertexStyle` option to a single style for global changes or to a list of rules to set options for individual vertices as is shown below.

```
In[65]:= HypercubeGraph[2, VertexLabels → "Name",
          VertexStyle → {1 → Red, 2 → Green, 3 → Blue, 4 → Black},
          ImagePadding → 10]
```

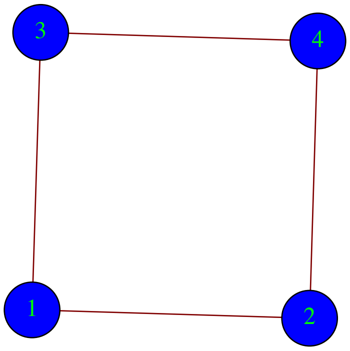
Out[65]=



The `VertexStyle` option is not a possibility for graphs not stored as a `Graph` object. Instead, the `VertexRenderingFunction` allows us to take complete control of the appearance of vertices. It is not quite so easy to use, however. The following shows how to display the vertices as blue circles with the vertex label inside of the disk.

```
In[66]:= GraphPlot[{1 → 2, 1 → 3, 2 → 4, 3 → 4}, VertexRenderingFunction →
            ({Blue, EdgeForm[Black], Disk[#1, .1], Green, Text[#2, #1]} &)]
```

Out[66]=

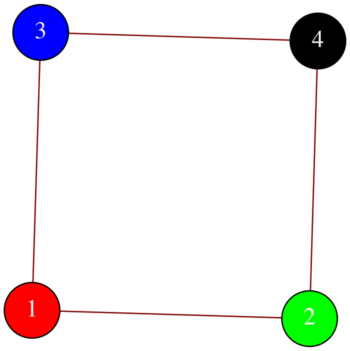


Notice that the `VertexRenderingFunction` is set to a pure `Function` (&). Each time a vertex needs to be drawn, this function is called with two arguments: the location of the center of the vertex (**#1**) and the name of the vertex (**#2**). The body of the function is a list of graphics directives. **Blue** and **EdgeForm[Black]** cause the shape to be filled with blue and to have a black border. The shape **Disk[#1,.1]** causes a disk to be drawn at the location of the vertex (**#1**) with radius .1. Then **Green** changes the prevailing color and **Text[#2,#1]** causes the name of the vertex (**#2**) to be displayed at the location (**#1**) of the vertex. The interested reader can explore the help page for `VertexRenderingFunction` for more information.

Elements of the `VertexRenderingFunction` can be made dependent on particular vertices by using expressions such as `If` or `Switch`. For example, the following will change the color for each vertex.

```
In[67]:= GraphPlot[{1 → 2, 1 → 3, 2 → 4, 3 → 4}, VertexRenderingFunction →
           ({Switch[#2, 1, Red, 2, Green, 3, Blue, 4, Black],
              EdgeForm[Black], Disk[#1, .1], White, Text[#2, #1]} &)]
```

Out[67]=



We now turn to our version of `BipartiteGraphQ`. The idea of the function is as follows. (Note that this method is based on forming a spanning tree of the graph, a concept discussed in Section 11.4 of the textbook).

1. Pick a vertex *v* from the vertex set and place it in the set *A*.
2. Place all of *v*'s neighbors in set *B*.
3. For each vertex *w* in the set *B* that has not already been processed, place all of *w*'s neighbors that are not already in either set into the set *A*.
4. Repeat step 3, reversing *A* and *B* until no more vertices remain to be processed.
5. Once step 4 is complete, we have formed a disjoint partition of the vertices. We then examine each edge of the graph and ensure that no edge has both ends in *A* or both ends in *B*. If some edge fails that test, then the graph is not bipartite. If all of the edges do pass the test, then the graph is bipartite and (*A*, *B*) is a bipartition.

First we will need a function to determine the list of neighbors of a given vertex. For a `Graph` object, the built-in function `AdjacencyList` applied to a graph and a vertex will return the list of vertices adjacent to it.

To find the neighbors of a vertex in a graph defined by a list of rules, rather than a `Graph` object, we need to find all of the edges containing the given vertex and output the other vertex in the rule. The `Cases` function can be used for this. The first argument to `Cases` is a list of expressions. The second argument is a pattern expressing which elements of the first argument should be output. In our case, we want to pick out those edges, that is, `Rule`s, one of whose elements is the desired vertex. As an example, suppose we're looking for edges involving the vertex *a*. Those edges will either be of the form **`Rule["a",_]`** or **`Rule[_,"a"]`**. In a pattern, we use the `Alternatives` (|) operator to combined two or more possibilities in a single pattern. The following picks out all of the edges incident to *a* in **`exercise2edges`**.

```
In[68]:= exercise2edges

Out[68]= {a → a, a → b, a → b, a → b, a → e, b → c,
          b → d, b → e, c → c, c → d, c → d, c → d, d → e}
```

```
In[69]:= Cases[exercise2edges, Rule["a", _] | Rule[_, "a"]]

Out[69]= {a → a, a → b, a → b, a → b, a → e}
```

Note that these edges form the neighborhood graph of “a”, that is, the subgraph consisting of “a” and all of its neighbors. But our interest is the neighbors, not the subgraph. To obtain the neighbors, we’ll take advantage of another feature of the `Cases` function: the second argument can be given as a rule. The left hand side of the rule is the pattern expressing which elements of the original list should match and the right hand side describes what to include in the output for that matching element. Since we want the other vertex in the output, not the entire rule, we’ll name the blanks in the rule and output the vertex.

```
In[70]:= Cases[exercise2edges, Rule["a", x_] | Rule[x_, "a"] → x]

Out[70]= {a, b, b, b, e}
```

Applying `Union` will remove duplicates and sort the results. Replacing the example data with arguments allows us to create a function.

```
In[71]:= neighbors[E : {__Rule}, v_] :=
          Module[{x}, Union[Cases[E, Rule[v, x_] | Rule[x_, v] → x]]]
```

Here is the implementation of our function **drawBipartite**.

```
In[72]:= drawBipartite[E : {__Rule}] := Module[{V, AB, i, T, w, e},
           V = Union[Flatten[E, 2, Rule]];
           w = V[[1]];
           AB[0] = {w};
           AB[1] = {};
           i = 0;
           While[V ≠ {},
            T = Intersection[V, AB[i]];
            i = Mod[i + 1, 2];
            Do[AB[i] = Union[AB[i],
               Complement[neighbors[E, w], Union[AB[0], AB[1]]]]
              , {w, T}];
            V = Complement[V, T]
           ];
           Catch[
            Do[If[(MemberQ[AB[0], e[[1]]] && MemberQ[AB[0], e[[2]]]) ||
                (MemberQ[AB[1], e[[1]]] && MemberQ[AB[1], e[[2]]]),
               Throw[False]]
              , {e, E}];
            GraphPlot[E, VertexRenderingFunction →
               ({If[MemberQ[AB[0], #2], Red, Green], EdgeForm[Black],
                  Disk[#1, .1], Black, Text[#2, #1]} &)]
           ]
          ]
```

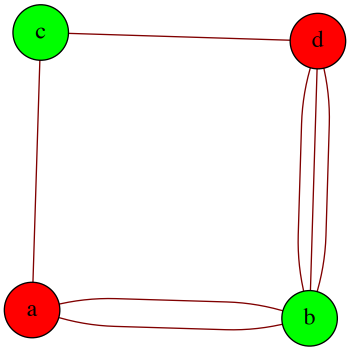
Note that if the graph is not bipartite, the function will output false.

```
In[73]:= drawBipartite[exercise2edges]

Out[73]= False
```

But for bipartite graphs, such as Exercise 4 from Section 10.1, the function will draw the graph with the vertices colored to illustrate the bipartition.

```
In[74]:= drawBipartite[exercise4edges]
```

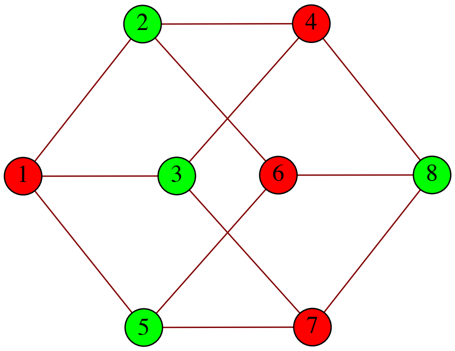
Out[74]=



Note also that our function can be applied to a Graph object by calling EdgeList. Then, since EdgeList always returns a list whose elements have head UndirectedEdge or DirectedEdge, we must transform those heads into Rule. For example, the following illustrates a bipartition for the three dimensional hypercube.

```
In[75]:= drawBipartite[
          EdgeList[HypercubeGraph[3]] /. UndirectedEdge → Rule]
```

Out[75]=



For convenience, we can define a version of **drawBipartite** that accepts a Graph object argument and applies the transformation automatically.

```
In[76]:= drawBipartite[G_Graph] := drawBipartite[
           EdgeList[G] /. {DirectedEdge → Rule, UndirectedEdge → Rule}
          ]
```

Bipartite Graphs and Matchings

Mathematica can help us find maximal matchings in a bipartite graph. We will use Figure 10a from Section 10.2 of the text as an example. To improve readability, we have abbreviated the names to their first letter and shortened the descriptions of the jobs.

```
In[77]:= figure10aEdges = {"A" → "req", "A" → "test",
           "B" → "arch", "B" → "imp", "B" → "test",
           "C" → "req", "C" → "arch", "C" → "imp", "D" → "req"}

Out[77]= {A → req, A → test, B → arch, B → imp,
          B → test, C → req, C → arch, C → imp, D → req}
```

In order to draw the graph meaningfully, in the same fashion as in the text, we will specify the coordinates of each vertex. This is done by setting the `VertexCoordinates` option to a list of coordinates, with the order of the list matching the order of the vertices. To ensure that our order is correct, we will specify a vertex list when we create the `Graph`.

```
In[78]:= figure10aVertices =
          {"A", "B", "C", "D", "req", "test", "arch", "imp"}

Out[78]= {A, B, C, D, req, test, arch, imp}
```

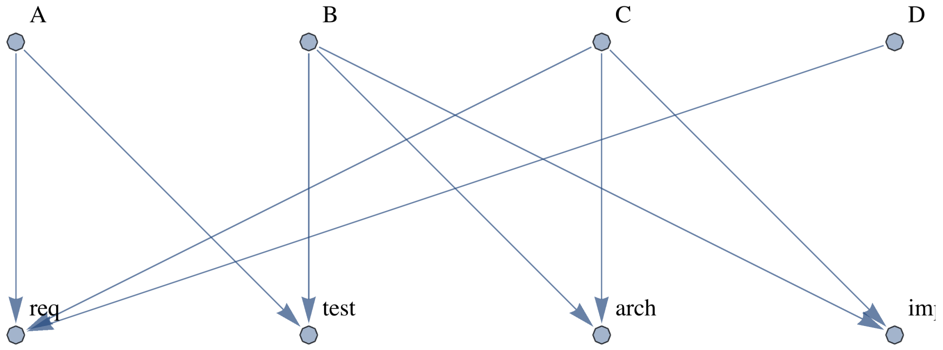
To create the list of coordinates, we use two `Table`s and specify that the names should have y-coordinate 1 and the jobs should have y-coordinate 0. `Join` is used to combine the two lists.

```
In[79]:= figure10aCoordinates =
          Join[Table[{i, 1}, {i, 4}], Table[{i, 0}, {i, 4}]]

Out[79]= {{1, 1}, {2, 1}, {3, 1}, {4, 1}, {1, 0}, {2, 0}, {3, 0}, {4, 0}}
```

We now create the graph.

```
In[80]:= figure10a = Graph[figure10aVertices,
           figure10aEdges, VertexLabels → "Name",
           VertexCoordinates → figure10aCoordinates, ImagePadding → 5]
```

Out[80]=



To find a maximal matching, we use the function `FindIndependentEdgeSet`. The term *independent edge set* is synonymous with matching. The only allowed argument to this function is the graph. It returns a list of edges in a matching.

In[81]:=
```
figure10aMatching = FindIndependentEdgeSet[figure10a]
```

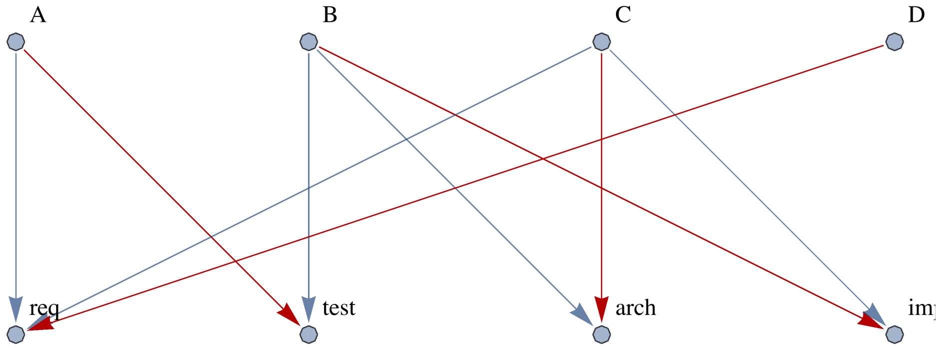
Out[81]=
```
{A → test, B → imp, C → arch, D → req}
```

The output indicates that one maximal matching has Alvarez assigned to testing, Berkowitz to implementation, Chen to architecture, and Davis to requirements.

We can visualize this matching by having *Mathematica* highlight the edges that form the matching using the function `HighlightGraph`. This function requires two arguments. The first is a graph. The second is a list of the elements to highlight. In this case, the second argument will be the output from `FindIndependentEdgeSet`.

In[82]:=
```
HighlightGraph[figure10a, figure10aMatching]
```

Out[82]=



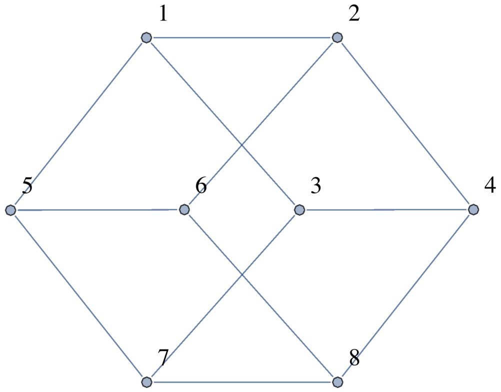
Subgraphs and Induced Subgraphs

Mathematica provides the `Subgraph` function for producing a subgraph from an existing graph. Given a graph and a list of edges, `Subgraph` produces the `Graph` consisting of the edges and all the vertices that are an endpoint of one of the given edges.

For example, below we create the subgraph of the hypercube graph consisting of one of the faces of the cube.

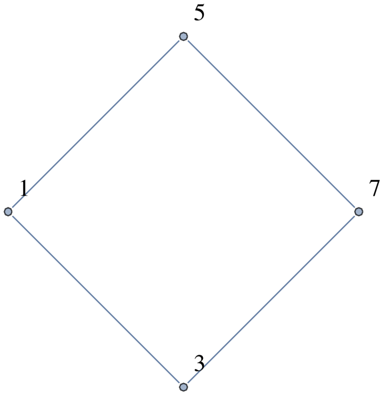
In[83]:=
```
hyper =
  HypercubeGraph[3, VertexLabels → "Name", ImagePadding → 10]
```

Out[83]=



```
In[84]:= subhyper = Subgraph[hyper, {1 → 3, 3 → 7, 7 → 5, 5 → 1},
           VertexLabels → "Name", ImagePadding → 10]
```

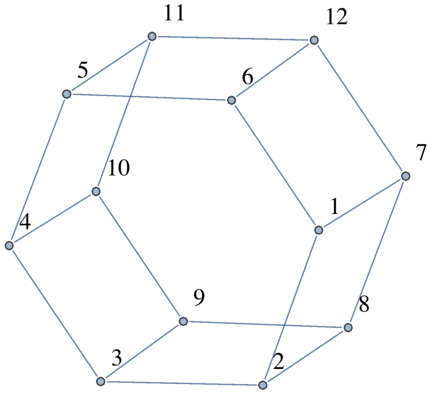
Out[84]=



Alternately, you can give a list of vertices as the second argument to `Subgraph`. The result is the graph induced by the given vertices, that is, the graph consisting of the vertices and all the edges from the original graph with both endpoints in the set of vertices. Below we consider a prism graph and one of its layers.

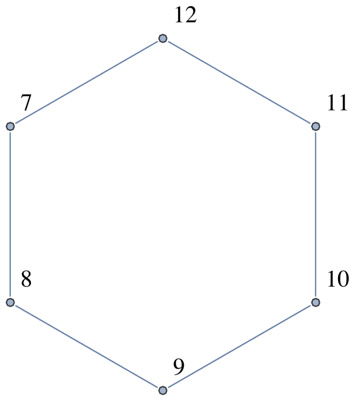
```
In[85]:= prism = Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 1, 7 → 8,
             8 → 9, 9 → 10, 10 → 11, 11 → 12, 12 → 7, 1 → 7, 2 → 8,
             3 → 9, 4 → 10, 5 → 11, 6 → 12}, DirectedEdges → False,
           VertexLabels → "Name", ImagePadding → 10]
```

Out[85]=



```
In[86]:= subprism = Subgraph[prism, {7, 8, 9, 10, 11, 12},
           VertexLabels → "Name", ImagePadding → 10]
```

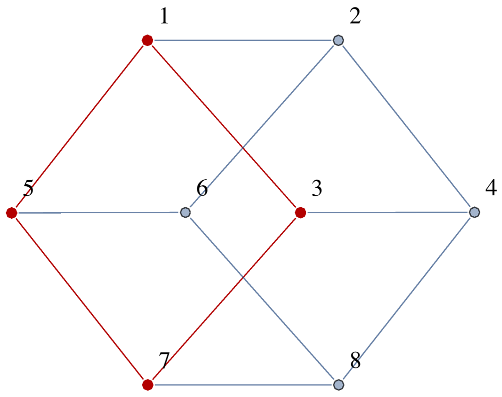
Out[86]=



The `HighlightGraph` function can be used to display a subgraph relative to the original by passing the original graph as the first argument and the subgraph as the second.

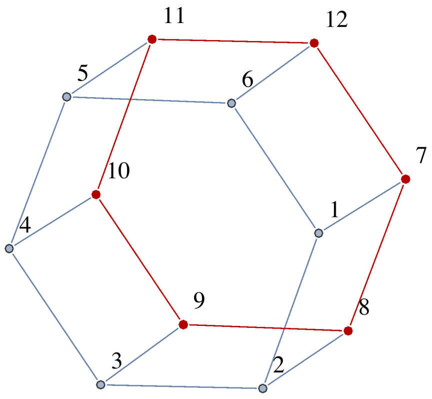
```
In[87]:= HighlightGraph[hyper, subhyper]
```

Out[87]=



```
In[88]:= HighlightGraph[prism, subprism]
```

Out[88]=

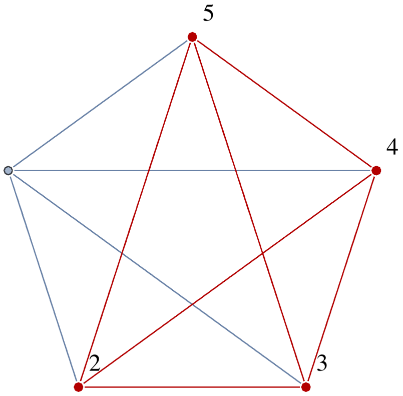


Deleting Vertices and Edges

Subgraphs can also be produced by deleting vertices or edges. The `VertexDelete` and `EdgeDelete` functions were described in the previous section, but are worth revisiting. `VertexDelete` takes two arguments: a graph and a vertex or list of vertices. The function returns a new graph with the vertex or vertices and all incident edges removed. Here we highlight the subgraph of the complete graph K_4 that is obtained by deleting a vertex.

```
In[89]:= deleteVExStart = CompleteGraph[5];

In[90]:= deleteVExEnd = VertexDelete[deleteVExStart, 1];

In[91]:= HighlightGraph[deleteVExStart, deleteVExEnd,
          VertexLabels → "Name", ImagePadding → 10]
```

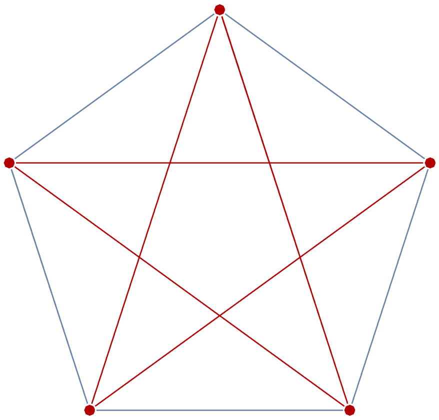
Out[91]=



`EdgeDelete` also takes two arguments, a name of an undirected graph and an edge or a list of edges. For example, we can remove the outer ring of K_5 as follows.

```
In[92]:= deleteEexStart = CompleteGraph[5];

In[93]:= deleteEexEdges = Join[Table[i → i + 1, {i, 1, 4}], {1 → 5}];

In[94]:= deleteEexEnd = EdgeDelete[deleteEexStart, deleteEexEdges];

In[95]:= HighlightGraph[deleteEexStart, deleteEexEnd]
```

Out[95]=

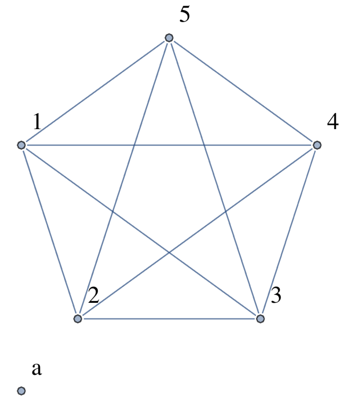


Adding Vertices and Edges

The functions for adding vertices and edges are very similar. `VertexAdd` accepts a graph and either a vertex or list of vertices to add to the graph.

```
In[96]:= VertexAdd[CompleteGraph[5,
           VertexLabels → "Name", ImagePadding → 10], "a"]
```

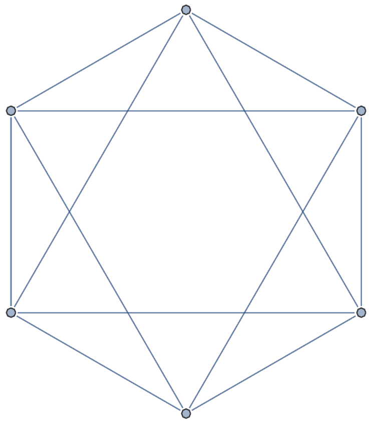
Out[96]=



`EdgeAdd` acts on a graph and adds an edge or a list of edges. Note that you can use rules to describe the edges and *Mathematica* will interpret them as directed or not depending on whether the original graph is directed.

```
In[97]:= EdgeAdd[CycleGraph[6], {1 → 3, 2 → 4, 3 → 5, 4 → 6, 5 → 1, 6 → 2}]
```

Out[97]=

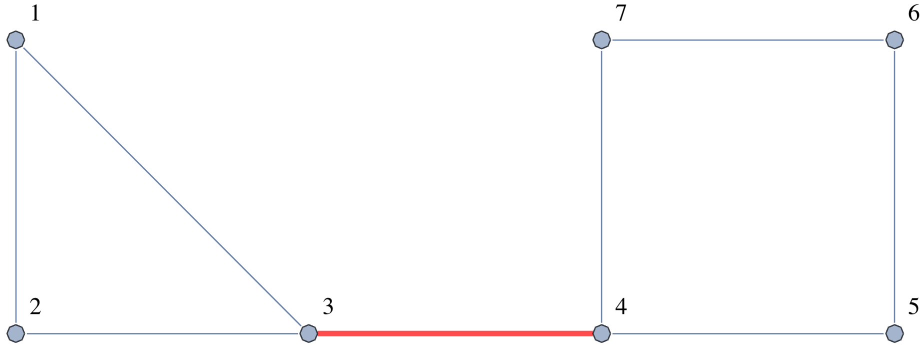


Edge Contraction

Recall that an edge contraction for an edge *e* with endpoints *u* and *v* consists of deleting the edge, merging *u* and *v* into a new vertex *w*, and preserving all edges (other than *e*) which had *u* or *v* as an endpoint by setting *w* as the new endpoint. As an illustration, consider the following graph.

```
In[98]:= exampleContraction =
          Graph[{1, 2, 3, 4, 5, 6, 7}, {1 → 2, 1 → 3, 2 → 3,
            Style[3 → 4, {Thick, Red}], 4 → 5, 4 → 7, 5 → 6, 6 → 7},
           DirectedEdges → False, VertexLabels → "Name",
           ImagePadding → 5, VertexCoordinates →
            {{0, 1}, {0, 0}, {1, 0}, {2, 0}, {3, 0}, {3, 1}, {2, 1}}]
```

Out[98]=



Observe the use of the `Style` wrapper on the edge between vertices 3 and 4 to highlight that edge.

Mathematica does not include a built-in function for performing an edge contraction, so we will create one. Our function will take as arguments a `Graph` object and a rule or a directed edge or an undirected edge representing the edge to be contracted. Since some of the `Graph` functions may cause an error if given an undirected graph and a rule, we begin the function by ensuring that if the graph is undirected then the edge is stored as an `UndirectedEdge` and a `DirectedEdge` if not.

After ensuring the edge is represented properly, we execute a `Do` loop over the vertices **x** of the original graph. This loop sets values of an indexed variable **vertLocs**, which stores the locations of the vertices from the display of the original graph. This will be used later to ensure that the vertex positions are preserved in the output. The positions are obtained using the function `PropertyValue` which takes two arguments: a list consisting of the graph and an object (vertex or edge) of the graph, and the name of a property, in this case, `VertexCoordinates`. Doing this is not necessary for performing the contraction, but then *Mathematica* would choose new positions for the vertices in the output graph, which may make the connection between the two graphs more difficult to see.

Next, the function deletes the edge being contracted.

We will need to create a new vertex. The function will automatically name this new vertex by combining the names of the original two vertices with a hyphen, as in 3-4. To do this, we use `ToString` on each of the original vertex names to ensure that they are string, and then combine them with the hyphen using the `StringJoin` (**<>**) operator. This vertex is then added to the graph and its position is calculated as the average of the positions of the original vertices.

Once the new vertex has been established, it needs to be connected to the neighbors of the original vertices. To do this, we use `NeighborhoodGraph` applied to the existing graph and the endpoints of the removed edge. The result of this function is the graph consisting of those two vertices and all of their neighbors. The edges of this graph are exactly the edges that need to be modified to create the contraction. Specifically, for each edge in the neighborhood graph, we need to replace u or v with the new vertex w.

To this end, we loop over all of the edges in the neighborhood graph, with loop variable **`modEdge`**. First, the edge is deleted from the existing graph. Then we modify the edge by applying `ReplaceAll` (**`/.`**) to replace whichever of the contracted vertices are present with the new vertex. Provided this edge does not already exist, we add it to the graph.

Once the edges have been modified, the two vertices are deleted from the graph. And finally, looping over the vertices of the graph, we use `PropertyValue`, to assign the original positions, which were stored in **`vertLocs`**.

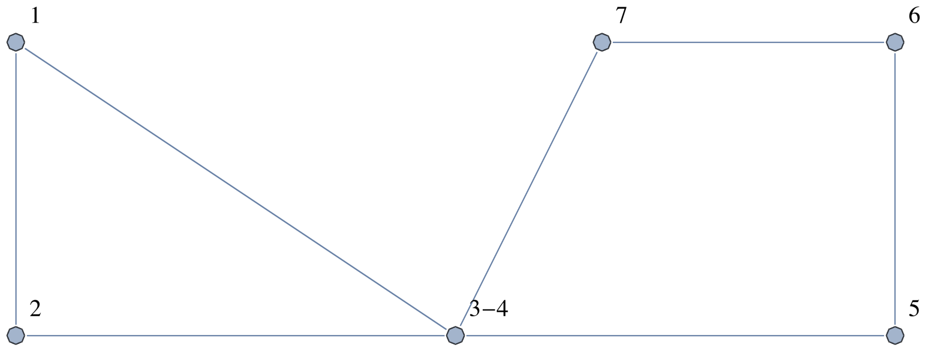
Here is the function.

```
In[99]:= contractGraph[G_Graph,
           e : _Rule | _DirectedEdge | _UndirectedEdge] :=
          Module[{u = e[[1]], v = e[[2]], contractE,
            vertLocs, x, w, g = G, N, modEdge},
           (* ensure the edge is a DirectedEdge or UndirectedEdge *)
           If[UndirectedGraphQ[G], contractE = UndirectedEdge[u, v],
            contractE = DirectedEdge[u, v]];
           (* store vertex locations *)
           Do[vertLocs[x] = PropertyValue[{G, x}, VertexCoordinates],
            {x, VertexList[G]}];
           (* delete the edge and add the new vertex *)
           g = EdgeDelete[g, contractE];
           w = ToString[u] <> "-" <> ToString[v];
           g = VertexAdd[g, w];
           vertLocs[w] = (vertLocs[u] + vertLocs[v]) / 2;
           (* update old edges *)
           N = NeighborhoodGraph[g, {u, v}];
           Do[g = EdgeDelete[g, modEdge];
            modEdge = modEdge /. {u → w, v → w};
            If[! EdgeQ[g, modEdge], g = EdgeAdd[g, modEdge]]
             , {modEdge, EdgeList[N]}];
           (* delete the old vertices and reset vertex positions *)
           g = VertexDelete[g, {u, v}];
           Do[PropertyValue[{g, x}, VertexCoordinates] = vertLocs[x],
            {x, VertexList[g]}];
           g
          ]
```

We apply **`contractGraph`** to the example.

```
In[100]:= contractGraph[exampleContraction, 3 → 4]
```

Out[100]=



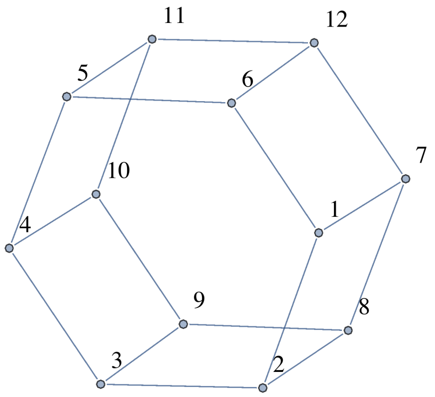
Unions and Complements of Graphs

Recall that the union of two graphs is the graph obtained by taking the union of the sets of vertices and the sets of edges from the two graphs.

As an example, we will “fill in” a prism graph by computing the union of the prism with the complete graph on the vertices in one ring. We begin with the prism we created above.

```
In[101]:= unionExampleA = prism
```

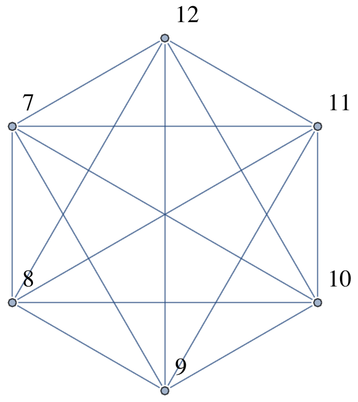
Out[101]=



We use the complete graph on 6 vertices as the second graph that will form part of the union. By default, `CompleteGraph` with argument 6 will form the complete graph on the vertices from 1 through 6. Suppose instead that we want the complete graph on the vertices from 7 through 12. To do this, apply the `VertexReplace` function. This function accepts two arguments: a graph and a list of rules specifying how to modify the names of the vertices. In this instance, we will replace vertex 1 with 7, 2 with 8, and so on, so we need to use the list of rules {1 → 7, 2 → 6, …, 6 → 12}, which we will create with a `Table`.

```
In[102]:= unionExampleB =
            VertexReplace[CompleteGraph[6, VertexLabels → "Name",
              ImagePadding → 10], Table[i → i + 6, {i, 6}]]
```

Out[102]=

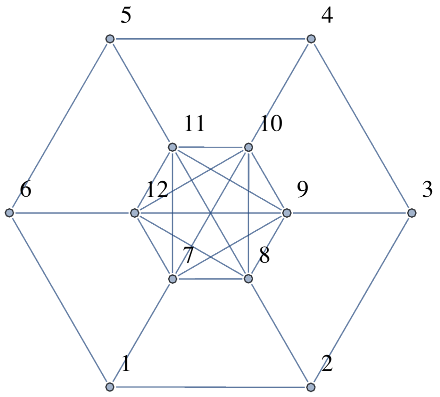


Note that the same effect can be obtained with the function `IndexGraph`, with first argument a graph and second argument the smallest integer to be used (defaulting to 1 if the second argument is omitted). `VertexReplace` is the more general function.

To obtain the union of the graphs, we apply the `GraphUnion` function, which simply takes the two (or more) graphs as arguments, along with the usual options.

```
In[103]:= unionExample = GraphUnion[unionExampleA,
            unionExampleB, VertexLabels → "Name", ImagePadding → 10]
```

Out[103]=

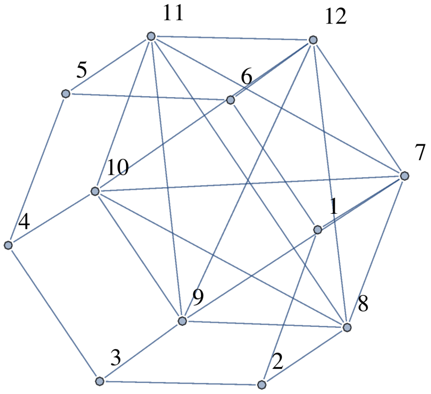


Note that *Mathematica* has rearranged the locations of the vertices. If you prefer the three-dimensional appearance of the prism, you can impose those locations as shown below. Note how the `PropertyValue` function is being used to both access the values from **unionExampleA** and assign those locations to **unionExample**.

```
In[104]:= Do[PropertyValue[{unionExample, v}, VertexCoordinates] =
            PropertyValue[{unionExampleA, v}, VertexCoordinates],
           {v, VertexList[unionExample]}]
```

```
In[105]:= unionExample
```

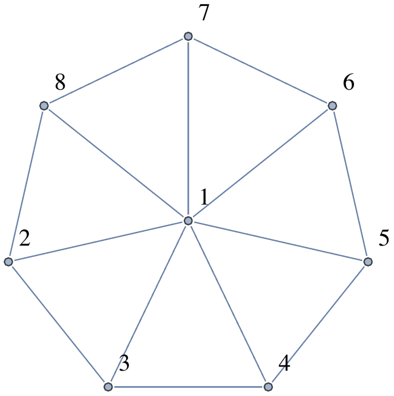
Out[105]=



Finally, we consider graph complements, described in Exercise 59 of Section 10.2. The complement, $\overline{G}$, of a graph G is the graph whose vertex set is the same as that of G, but whose edge set is the set of all pairs of G that have no edge between them. In other words, if G has n vertices, then the edge set of G is the complement of the edge set of G relative to K_n, the complete graph on n vertices. *Mathematica* has a function to compute the complement of a graph: `GraphComplement`.

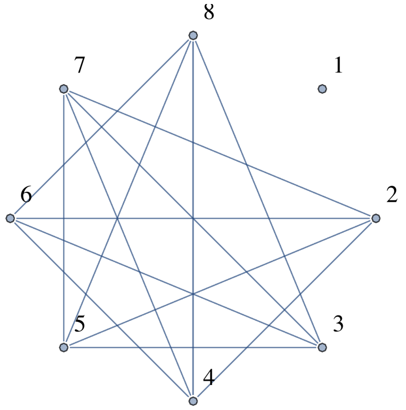
```
In[106]:= WheelGraph[8, VertexLabels → "Name", ImagePadding → 10]
```

Out[106]=



```
In[107]:= complementExample = GraphComplement[
            WheelGraph[8], VertexLabels → "Name", ImagePadding → 7]
```

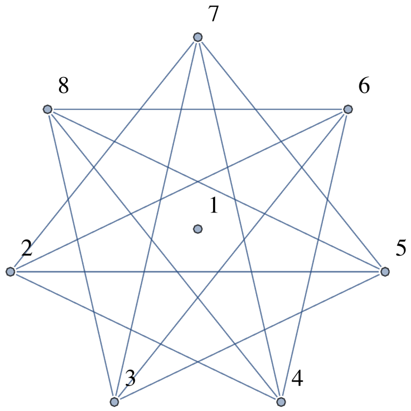
Out[107]=



Again, *Mathematica* has rearranged the vertices. We can impose the original locations as follows.

```
In[108]:= Do[PropertyValue[{complementExample, v}, VertexCoordinates] =
            PropertyValue[{WheelGraph[8], v}, VertexCoordinates],
          {v, VertexList[complementExample]}]

In[109]:= complementExample
```

Out[109]=



10.3 Representing Graphs and Graph Isomorphism

In this section we will see how to represent graphs in terms of adjacency matrices, adjacency lists, and incidence matrices. We will then use the adjacency matrix representation to help determine whether two graphs are isomorphic.

Adjacency Matrices

The adjacency matrix of a graph G with n vertices is the $n \times n$ matrix whose (i, j) entry is 1 if there is an edge from vertex i to vertex j and 0 if not. You can define a graph by passing an adjacency matrix, represented as a list of lists, to the function `AdjacencyGraph`.

As an example, we reproduce Example 4 from Section 10.3.

```
In[110]:= exampleAdjM =
            {{0, 1, 1, 0}, {1, 0, 0, 1}, {1, 0, 0, 1}, {0, 1, 1, 0}};
          exampleAdjM // MatrixForm
```

Out[111]//MatrixForm=

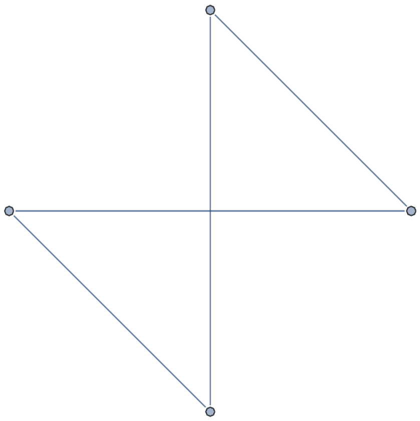
$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Recall that we must invoke `MatrixForm` in an expression separate from the definition of the symbol in order to avoid having the `MatrixForm` head permanently stored in the symbol.

We now invoke the `AdjacencyGraph` function with this matrix.

```
In[112]:= AdjacencyGraph[exampleAdjM]
```

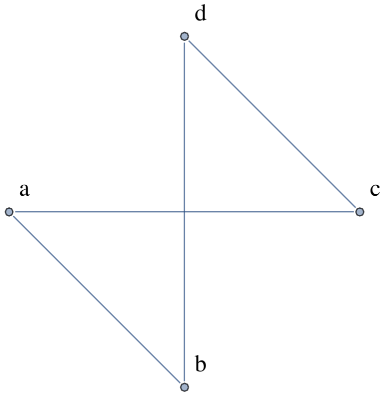
Out[112]=



In the textbook, the vertices for this graph were labeled as letters rather than numbers. The `AdjacencyGraph` function accepts a list of names for the vertices as an optional first argument. It also accepts the usual graph options.

```
In[113]:= AdjacencyGraph[{"a", "b", "c", "d"}, exampleAdjM,
           VertexLabels → "Name", ImagePadding → 10]
```

Out[113]=



Notice that this is the same graph as is produced in the textbook, with the exception of the locations of the vertices.

Mathematica also provides a function, `AdjacencyMatrix`, for computing the adjacency matrix of a graph. The output of this function is always a `SparseArray` object, which is more efficient at storing large arrays with many entries 0. You can display a `SparseArray` with `MatrixForm`, as usual, and you can convert it into the usual list of lists representation with the function `Normal`.

```
In[114]:= wheelAdjacency = AdjacencyMatrix[WheelGraph[7]]

Out[114]= SparseArray[<24>, {7, 7}]

In[115]:= wheelAdjacency // MatrixForm

Out[115]//MatrixForm=
          ( 0 1 1 1 1 1 1 )
          ( 1 0 1 0 0 0 1 )
          ( 1 1 0 1 0 0 0 )
          ( 1 0 1 0 1 0 0 )
          ( 1 0 0 1 0 1 0 )
          ( 1 0 0 0 1 0 1 )
          ( 1 1 0 0 0 1 0 )

In[116]:= wheelAdjacency // Normal

Out[116]= {{0, 1, 1, 1, 1, 1, 1}, {1, 0, 1, 0, 0, 0, 1},
           {1, 1, 0, 1, 0, 0, 0}, {1, 0, 1, 0, 1, 0, 0},
           {1, 0, 0, 1, 0, 1, 0}, {1, 0, 0, 0, 1, 0, 1}, {1, 1, 0, 0, 0, 1, 0}}
```

Adjacency Lists

Recall that a representation of a graph as an adjacency list consists of the lists of neighbors of each vertex.

Above, we saw the function `AdjacencyList` used to determine the list of vertices adjacent to a given vertex. For example, the following determines the vertices adjacent to vertex 2 in the wheel graph on 7 vertices.

```
In[117]:= AdjacencyList[WheelGraph[7], 2]

Out[117]= {1, 3, 7}
```

To obtain the complete adjacency list representation of a graph, it is only a matter of looping through the vertices of the graph.

```
In[118]:= adjacencyList[G_Graph] :=
           Table[AdjacencyList[G, v], {v, VertexList[G]}]

In[119]:= adjacencyList[WheelGraph[7]]

Out[119]= {{2, 3, 4, 5, 6, 7}, {1, 3, 7}, {1, 2, 4},
           {1, 3, 5}, {1, 4, 6}, {1, 5, 7}, {1, 2, 6}}
```

Mathematica does not include a function to create a graph from an adjacency list. However, it is not difficult to create a function that transforms an adjacency list into a graph object by using the adjacency matrix as an intermediate. We will create a function **adjacencyListGraph** that accepts as its argument a list of lists. We will require that the vertices be represented by positive integers beginning with 1. For example, {{2, 3}, {1, 3, 4}, {1, 2}, {2, 5}, {4}} will be used to represent the adjacency list for a graph in which vertex 1 is incident to vertices 2 and 3; vertex 2 is incident to vertices 1, 3, and 4; vertex 3 is incident to vertices 1 and 2; and so on.

The main work of the function will be to transform the list into a matrix with 1s in the locations specified by the adjacency list. The first sublist in the adjacency list indicates the positions in the first row of the adjacency matrix that should be set to 1. In the example, {2, 3} tells us that the first row should have 1s in the second and third columns. The second sublist specifies the second row, and so on.

We mentioned above that a `SparseArray` is particularly suitable for matrices with few non-zero entries. To further explore this type of object, we will have our **adjacencyListGraph** function create a `SparseArray` as part of its operation. There are several different syntax options for creating a `SparseArray`, but we will use the most descriptive: a list of rules identifying positions with values. For example, to create a 4×4 matrix is 1s in positions (1, 3), (2, 4), and (4, 1), you enter the following.

```
In[120]:= SparseArray[{{1, 3} → 1, {2, 4} → 1, {4, 1} → 1}, {4, 4}] //
           MatrixForm
```

Out[120]//MatrixForm=

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

Observe that the first argument is a list of rules with the positions within the matrix given as lists on the left hand of each rule, and the value that belongs in the position on the right. The second argument to `SparseArray` is a list specifying the dimensions of the matrix. If the second argument is omitted, *Mathematica* will attempt to deduce the size of the matrix from the given entries.

The **adjacencyListGraph** function will begin by determining the dimension of the matrix, which must be equal to the length of the adjacency list. The function must then create the list of rules from the adjacency list.

Recall that the `Map` function is used to apply a function to every member of a sequence. For example, the following can be used to add "*x*" to each element of a list.

```
In[121]:= Map[# + x &, {1, 2, 3, 4}]

Out[121]= {1 + x, 2 + x, 3 + x, 4 + x}
```

As you can see above, the function is applied to each element of the list with the `Slot` (#) replaced by the elements of the list. There is a related function, `MapIndexed`, whose first argument should be a function on two arguments, which are taken as the element of the list together with a specification of the location of that element within the list as the second. For example, the following applies the function "f" to a list of letters.

```
In[122]:= MapIndexed[f, {"a", "b", "c"}]

Out[122]= {f[a, {1}], f[b, {2}], f[c, {3}]}
```

You see that "f" is applied to two arguments: the letter from the second argument and a list containing the position of the letter in the list of letters. The position is given as a list in case of nested structures requiring more complicated position specifications. The following makes use of `MapIndexed` to add *x* raised to a power determined by the index of the list element. We use `First` to take the index out of the enclosing list.

```
In[123]:= MapIndexed[#1 + x^First[#2] &, {7, 3, 11, 4}]

Out[123]= {7 + x, 3 + x^2, 11 + x^3, 4 + x^4}
```

For nested lists, `MapIndexed` requires a third argument specifying the level at which to apply the function. For example, the following function applies the function "f" at the second level.

```
In[124]:= MapIndexed[f, {{1, 2}, {3, 4, 5}}, {2}]

Out[124]= {{f[1, {1, 1}], f[2, {1, 2}]},
           {f[3, {2, 1}], f[4, {2, 2}], f[5, {2, 3}]}}
```

Without the level **`{2}`**, which means that the function should be applied only to elements at the second level in the nested lists, **`f`** would have been applied only to the two lists {1, 2} and {3, 4, 5} instead of the five numbers 1 through 5. Note that the second arguments passed to f are lists consisting of two integers: the first indicating which sublist it is and the second indicating the position within the sublist.

If {{1, 2}, {3, 4, 5}} were an adjacency list, the first element from the index information tells us which row in the adjacency matrix that sublist refers to. The values in the lists tell us which columns in that row should be 1s. The following application of `MapIndexed` uses this observation to print out the locations of 1s in the adjacency matrix.

```
In[125]:= MapIndexed[Print[{First[#2], #1}] &, {{1, 2}, {3, 4, 5}}, {2}];

          {1, 1}

          {1, 2}

          {2, 3}

          {2, 4}

          {2, 5}
```

To form a `SparseArray`, we need to create a `Rule` (**`->`**) rather than printing the list. And we must apply `Flatten` since `MapIndexed` preserves the list structure and we need a simple list of rules.

```
In[126]:= Flatten[
           MapIndexed[{First[#2], #1} → 1 &, {{1, 2}, {3, 4, 5}}, {2}]
          ]

Out[126]= {{1, 1} → 1, {1, 2} → 1, {2, 3} → 1, {2, 4} → 1, {2, 5} → 1}
```

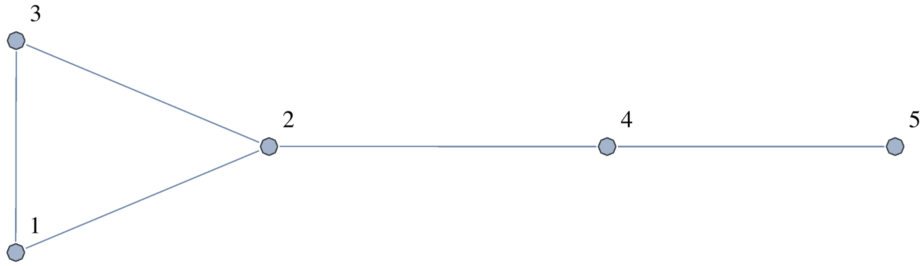
We now build **`adjacencyListGraph`**. We also illustrate how to use a `BlankNullSequence` (`___`), which matches any, including 0, number of arguments, to pass options from this function to `AdjacencyGraph`.

```
In[127]:= adjacencyListGraph[L_List, opts___] := Module[{n, rules},
            n = Length[L];
            rules =
             Flatten[MapIndexed[Rule[{First[#2], #1}, 1] &, L, {2}]];
            AdjacencyGraph[SparseArray[rules, {n, n}], opts]
           ]
```

We apply this to an example.

```
In[128]:= exampleAL =
           adjacencyListGraph[{{2, 3}, {1, 3, 4}, {1, 2}, {2, 5}, {4}},
            VertexLabels → "Name", ImagePadding → 5]
```

Out[128]=



Incidence Matrices

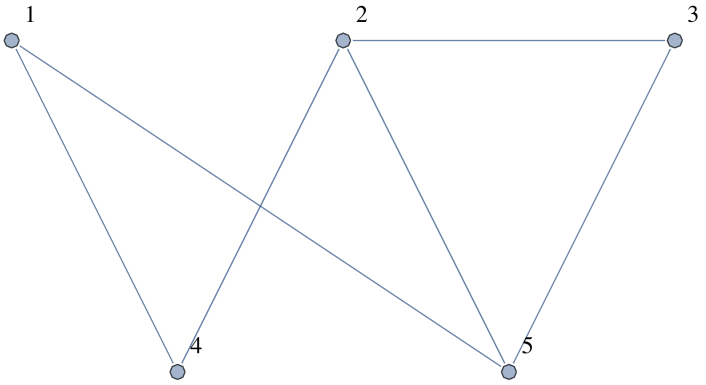
The third representation of graphs that we are considering is incidence matrices. For a graph G with n vertices and m edges, the associated incidence matrix is the $n \times m$ matrix whose (i, j) entry is 1 if vertex i is an endpoint of edge j.

Mathematica includes functions for working with incidence matrices. Given an incidence matrix M, the function `IncidenceGraph` will produce the associated graph. As an example, we reverse Example 6 from Section 10.3 and use the incidence matrix given in the solution in order to reproduce the graph.

```
In[129]:= exampleIncidenceM = {{1, 1, 0, 0, 0, 0}, {0, 0, 1, 1, 0, 1},
            {0, 0, 0, 0, 1, 1}, {1, 0, 1, 0, 0, 0}, {0, 1, 0, 1, 1, 0}}

Out[129]= {{1, 1, 0, 0, 0, 0}, {0, 0, 1, 1, 0, 1},
            {0, 0, 0, 0, 1, 1}, {1, 0, 1, 0, 0, 0}, {0, 1, 0, 1, 1, 0}}
```

```
In[130]:= exampleIncidenceG = IncidenceGraph[exampleIncidenceM,
            VertexLabels → "Name", ImagePadding → 10, VertexCoordinates →
             {{0, 1}, {1, 1}, {2, 1}, {0.5, 0}, {1.5, 0}}]
```

Out[130]=



For the reverse, the `IncidenceMatrix` function will produce the incidence matrix for a `Graph` object. We apply this function to the previous graph. Again, the output from this function is a `SparseArray` object, so we apply `MatrixForm` to view it.

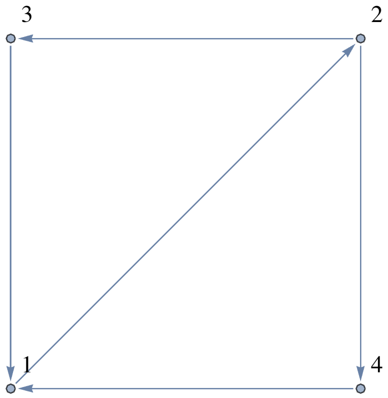
```
In[131]:= IncidenceMatrix[exampleIncidenceG] // MatrixForm
```

```
Out[131]//MatrixForm=
          1  1  0  0  0  0
          0  0  1  1  0  1
          0  0  0  0  1  1
          1  0  1  0  0  0
          0  1  0  1  1  0
```

For a directed graph, the `IncidenceMatrix` function returns a matrix with a 1 in position (i, j) indicating that the vertex i is the head of edge j and an entry of -1 indicating that the vertex is the tail of the edge.

```
In[132]:= directedIncidenceG = Graph[{1 → 2, 2 → 3, 3 → 1, 2 → 4, 4 → 1},
            DirectedEdges → True, VertexLabels → "Name", ImagePadding → 10,
            VertexCoordinates → {{0, 0}, {1, 1}, {0, 1}, {1, 0}}]
```

Out[132]=



```
In[133]:= IncidenceMatrix[directedIncidenceG] // MatrixForm

Out[133]//MatrixForm=
  -1   0   1   0   1
   1  -1   0  -1   0
   0   1  -1   0   0
   0   0   0   1  -1
```

Isomorphism of Graphs

We conclude this section with a brief discussion of isomorphism of graphs and graph invariants. Determining whether two graphs are isomorphic is a difficult problem. The naive approach (exhaustively checking each possible mapping) can require exponential time.

Graph invariants are useful tools for confirming that two graphs are not isomorphic. While there is no complete collection of graph invariants that will definitively conclude whether two graphs are or are not isomorphic, they can, for many pairs of graphs, quickly demonstrate the impossibility of an isomorphism. We will create a function that will check some of the basic invariants: number of vertices, number of edges, whether the graph is directed, and whether it is bipartite. We also introduce another invariant: the degree sequence.

For a graph G, the *degree sequence* is the list of the degrees of the vertices of the graph sorted in ascending order. The *Mathematica* function `VertexDegree` applied to a graph and a vertex returns the degree of the vertex. Applied to the graph with no second argument, it produces a list of the degrees of the vertices of a graph, listed in order of the vertices. Since this depends on the order in which *Mathematica* stores the vertices, it is not an invariant. However, applying the `Sort` function to the result of `VertexDegree` returns the degree sequence for the graph, which is an invariant.

The function defined below checks, one at a time, the invariants we have mentioned. If any of the invariants indicate that the graphs are not isomorphic, the procedure prints a statement to that effect.

```
In[134]:= checkInvariants[G1_Graph, G2_Graph] :=
           Module[{notIsomorphic = False},
            If[VertexCount[G1] ≠ VertexCount[G2],
             notIsomorphic = True;
             Print["Different numbers of vertices."]
            ];
            If[EdgeCount[G1] ≠ EdgeCount[G2],
               notIsomorphic = True;
               Print["Different numbers of edges."]
              ]'
             If[! Equivalent[DirectedGraphQ[G1], DirectedGraphQ[G2]],
              notIsomorphic = True;
              Print["One is directed, one is undirected."]
             ];
            If[! Equivalent[BipartiteGraphQ[G1], BipartiteGraphQ[G2]],
             notIsomorphic = True;
             Print["One is bipartite, one is not."]
            ];
            If[Sort[VertexDegree[G1]] ≠ Sort[VertexDegree[G2]],
             notIsomorphic = True;
             Print["Degree sequences do not match."]
            ];
            If[notIsomorphic,
             Print["The graphs are not isomorphic."],
             Print["The graphs MAY be isomorphic."]
            ]
           ]
```

```
In[135]:= checkInvariants[directedIncidenceG, exampleIncidenceG]
```

```
Different numbers of vertices.

Different numbers of edges.

One is directed, one is undirected.

Degree sequences do not match.

The graphs are not isomorphic.
```

```
In[136]:= checkInvariants[CompleteGraph[3], CycleGraph[3]]
```

```
The graphs MAY be isomorphic.
```

Mathematica provides a function, `IsomorphicGraphQ`, for definitively determining whether or not two graphs are isomorphic. This function applies to any `Graph` object. The `IsomorphicGraphQ` function accepts two graphs as its arguments. It returns `True` if the graphs are isomorphic.

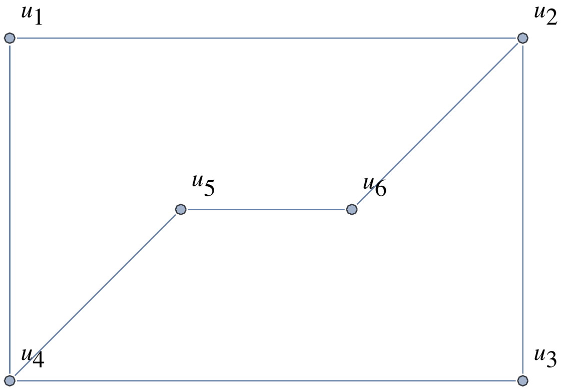
```
In[137]:= IsomorphicGraphQ[CompleteGraph[3], CycleGraph[3]]

Out[137]= True
```

The `FindGraphIsomorphism` function can be used to determine an explicit isomorphism for a pair of graphs that are in fact isomorphic. Like `IsomorphicGraphQ`, the only arguments are the two graphs. If the two graphs are in fact isomorphic, the output is a list of rules of the form $v \to w$ indicating that vertex v in the first graph is mapped to vertex w in the second graph. If the graphs are not isomorphic, `IsomorphicGraphQ` returns an empty list.

We illustrate by reproducing the graphs in Figure 12 of Section 10.3 of the textbook.

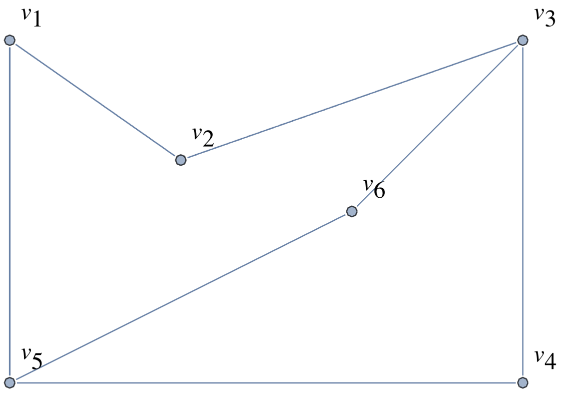
```
In[138]:= figure12G = Graph[{u1, u2, u3, u4, u5, u6},
            {u1 → u2, u1 → u4, u2 → u3, u2 → u6, u3 → u4, u4 → u5, u5 → u6},
            DirectedEdges → False, VertexLabels → "Name",
            ImagePadding → 10, VertexCoordinates →
             {{0, 2}, {3, 2}, {3, 0}, {0, 0}, {1, 1}, {2, 1}}]
```

Out[138]=



```
In[139]:= figure12H = Graph[{v1, v2, v3, v4, v5, v6},
            {v1 → v2, v1 → v5, v2 → v3, v3 → v4, v3 → v6, v4 → v5, v5 → v6},
            DirectedEdges → False, VertexLabels → "Name",
            ImagePadding → 10, VertexCoordinates →
             {{0, 2}, {1, 1.3}, {3, 2}, {3, 0}, {0, 0}, {2, 1}}]
```

Out[139]=



Applying IsomorphicGraphQ confirms that the graphs are isomorphic.

```
In[140]:= IsomorphicGraphQ[figure12G, figure12H]

Out[140]= True
```

FindGraphIsomorphism determines the isomorphism.

```
In[141]:= FindGraphIsomorphism[figure12G, figure12H]
```

Out[141]= $\{u_1 \to v_4, u_2 \to v_3, u_3 \to v_6, u_4 \to v_5, u_5 \to v_1, u_6 \to v_2\}$

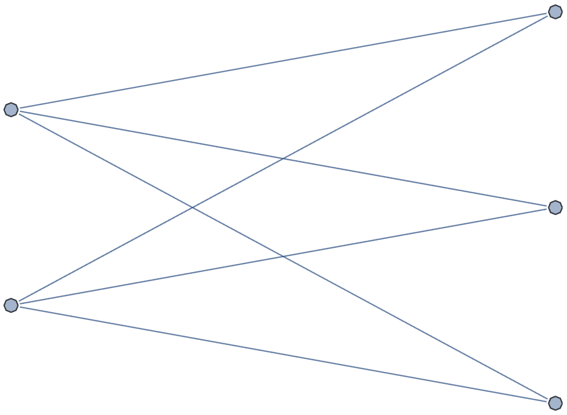
10.4 Connectivity

Mathematica provides a number of functions related to connectivity of graphs.

The first such function that we consider is ConnectedGraphQ. This function takes one argument, the name of the graph, and returns true or false. As an example, consider the complete bipartite graph $K_{2,3}$ and its complement.

```
In[142]:= CompleteGraph[{2, 3}]
```

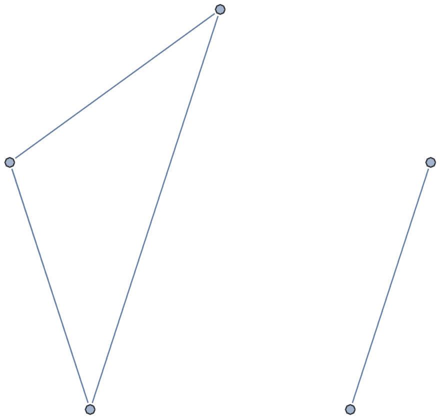
Out[142]=



```
In[143]:= ConnectedGraphQ[CompleteGraph[{2, 3}]]

Out[143]= True
```

```
In[144]:= GraphComplement[CompleteGraph[{2, 3}]]
```

Out[144]=

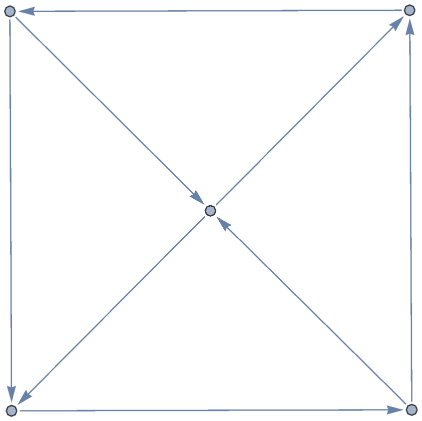


```
In[145]:= ConnectedGraphQ[GraphComplement[CompleteGraph[{2, 3}]]]

Out[145]= False
```

When used with a directed graph, ConnectedGraphQ returns True if the directed graph is strongly connected.

```
In[146]:= stronglyConnectedEx =
           Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 1 → 5, 5 → 2, 3 → 5, 5 → 4}]
```

Out[146]=



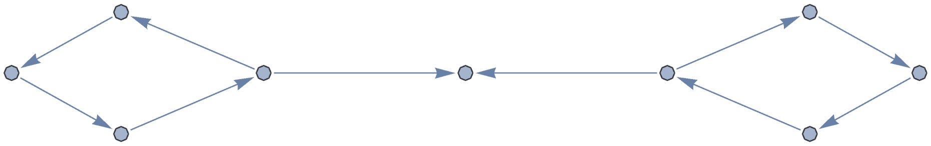
Applying ConnectedGraphQ reveals that the above graph is connected.

```
In[147]:= ConnectedGraphQ[stronglyConnectedEx]

Out[147]= True
```

The following example, while weakly connected, is not strongly connected and thus ConnectedGraphQ will return False.

```
In[148]:= weaklyConnectedEx = Graph[{4 → 2, 2 → 1,
              1 → 3, 3 → 4, 4 → 5, 6 → 8, 8 → 9, 9 → 7, 7 → 6, 6 → 5}]
```

Out[148]=



```
In[149]:= ConnectedGraphQ[weaklyConnectedEx]

Out[149]= False
```

The `WeaklyConnectedGraphQ` function will determine whether a directed graph is weakly connected. A graph is weakly connected if it is connected as an undirected graph.

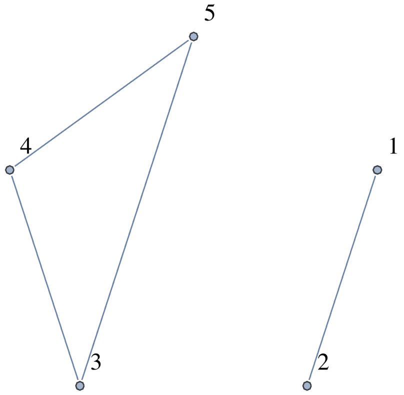
```
In[150]:= WeaklyConnectedGraphQ[weaklyConnectedEx]

Out[150]= True
```

Mathematica also has functions to extract the connected components of a graph that is not connected. The `ConnectedComponents` function takes a graph as input and returns a list of lists of vertices. For directed graphs, `ConnectedComponents` is used to determine the strongly connected components of the graph.

As an example, consider the complement of the graph $K_{2,3}$.

```
In[151]:= GraphComplement[CompleteGraph[{2, 3}],
           VertexLabels → "Name", ImagePadding → 10]
```

Out[151]=



```
In[152]:= ConnectedComponents[GraphComplement[CompleteGraph[{2, 3}]]]

Out[152]= {{3, 4, 5}, {1, 2}}
```

This output indicates that the complement of $K_{2,3}$ has two connected components, one with vertex set {1, 2} and the other with vertex set {3, 4, 5}.

For directed graphs, `ConnectedComponents` will produce the strongly connected components. The `WeaklyConnectedComponents` function will output the weakly connected components.

```
In[153]:= ConnectedComponents[weaklyConnectedEx]

Out[153]= {{5}, {4, 2, 1, 3}, {6, 8, 9, 7}}

In[154]:= WeaklyConnectedComponents[weaklyConnectedEx]

Out[154]= {{4, 2, 1, 3, 5, 6, 8, 9, 7}}
```

Coloring the Components

Now we present a function that will color code the connected components in a graph. (This function will color up to 5 components before repeating colors.)

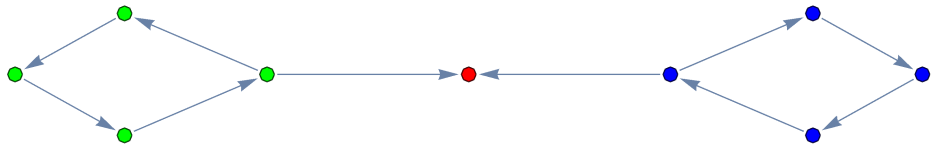
Note that *Mathematica* understands the symbols `Red`, `Green`, `Blue`, `Brown`, and `Gray` as Colors. Also note the use of `PropertyValue` to set the `VertexStyle` property. We first saw `PropertyValue` in the subsection on edge contraction in Section 10.2 of this manual. Finally, note the use of `Mod` with third argument 1, which causes the result of `Mod` to have minimum value 1. This allows us to cycle through the list of colors.

```
In[155]:= highlightComponents[G_Graph] :=
           Module[{colorList = {Red, Green, Blue, Brown, Gray},
             components, c, i, v, H = G},
            components = ConnectedComponents[G];
            c = 0;
            For[i = 1, i ≤ Length[components], i++,
             c = Mod[c + 1, 5, 1];
             Do[PropertyValue[{H, v}, VertexStyle] = colorList[[c]]
              , {v, components[[i]]}]
            ];
            H
           ]
```

We apply this function to the weakly connected graph above.

```
In[156]:= highlightComponents[weaklyConnectedEx]
```

Out[156]=



Counting Paths Between Vertices

The last topic that we consider in this section is determining the number of paths between two vertices of a given length. As described in the textbook, if A is the adjacency matrix for a graph (which may be undirected or directed and may include loops and multiple edges), then the (i, j) entry of the matrix A^r is the number of paths of length r from vertex i to vertex j.

As an example, consider the **`stronglyConnectedEx`** graph from above. We can obtain its adjacency matrix by applying the `AdjacencyMatrix` function to the name of the graph.

```
In[157]:= aMatrix = AdjacencyMatrix[stronglyConnectedEx];
          aMatrix // MatrixForm
```

```
Out[158]//MatrixForm=
 ( 0  1  0  0  1 )
 ( 0  0  1  0  0 )
 ( 0  0  0  1  1 )
 ( 1  0  0  0  0 )
 ( 0  1  0  1  0 )
```

You could use `Normal` to transform the `SparseArray` object into a usual list of lists representation of the matrix. However, *Mathematica* can compute more efficiently with the `SparseArray`.

Next, compute some powers of the adjacency matrix.

```
In[159]:= Table[MatrixForm[MatrixPower[aMatrix, i]], {i, 2, 7}]
```

```
            ( 0  1  1  1  0 )   ( 1  0  1  1  1 )   ( 1  2  0  2  2 )
            ( 0  0  0  1  1 )   ( 1  1  0  1  0 )   ( 1  1  1  0  1 )
Out[159]= { ( 1  1  0  1  0 ) , ( 1  1  1  0  1 ) , ( 0  2  1  2  2 ) ,
            ( 0  1  0  0  1 )   ( 0  1  1  1  0 )   ( 1  0  1  1  1 )
            ( 1  0  1  0  0 )   ( 0  1  0  1  2 )   ( 1  2  1  2  0 )

            ( 2  3  2  2  1 )   ( 2  3  3  3  4 )   ( 3  6  3  7  5 )
            ( 0  2  1  2  2 )   ( 2  2  2  3  1 )   ( 3  3  2  3  4 )
            ( 2  2  2  3  1 ) , ( 3  3  2  3  4 ) , ( 3  7  3  6  5 ) }
            ( 1  2  0  2  2 )   ( 2  3  2  2  1 )   ( 2  3  3  3  4 )
            ( 2  1  2  1  2 )   ( 1  4  1  4  4 )   ( 4  5  4  5  2 )
```

Note that the `MatrixPower` function is used to compute powers of matrices. Using the `Power` (^) operator on a matrix computes element-wise.

We see that there are 4 paths of length 6 from vertex 3 to vertex 5, since the (3, 5) entry in the 6th power of the adjacency matrix is 4. We also see that there are cycles of length 3 for every vertex and there are no cycles of length less than 3. Finally, we know that the shortest path from vertex 2 to vertex 1 is of length 3, since the (2, 1) entry is 0 for the first and second powers of the matrix.

10.5 Euler and Hamilton Paths

In this section we will show how to use *Mathematica* to solve two problems that seem closely related, but which are quite different in computational complexity. The two problems that will be analyzed are the problem of finding a simple circuit that contains every edge exactly once (an Euler circuit) and the problem of finding a simple circuit that contains every vertex exactly once (a Hamilton circuit). (Note that the textbook uses the term circuit while *Mathematica* uses the word cycle. These two terms are synonymous.)

Euler Circuits in Simple Graphs

Mathematica comes equipped with a function to determine if a given simple graph has an Euler circuit or not. This function, `EulerianGraphQ`, takes one argument, a `Graph` object. As an example, we'll have *Mathematica* check to see if K_5 is Eulerian, i.e., has an Euler circuit.

```
In[160]:= EulerianGraphQ[CompleteGraph[5]]

Out[160]= True
```

To explicitly find an Euler circuit, we use the function `FindEulerianCycle`. The function accepts one or two arguments. The first argument must be a graph, and if this is the only argument, the function returns a list containing a single list of edges representing an Euler circuit. For example, the following identifies an Euler circuit on the complete graph K_5.

```
In[161]:= FindEulerianCycle[CompleteGraph[5]]

Out[161]= {{1 ↔ 5, 5 ↔ 4, 4 ↔ 3, 3 ↔ 5,
             5 ↔ 2, 2 ↔ 4, 4 ↔ 1, 1 ↔ 3, 3 ↔ 2, 2 ↔ 1}}
```

If you provide `FindEulerianCycle` with a positive integer as a second argument, *Mathematica* will attempt to find more than one Euler circuit, with the integer serving as a maximum number of cycles to return, provided they exist. In this case, the output will be a list of lists of edges, with each sublist representing a distinct circuit.

```
In[162]:= FindEulerianCycle[CompleteGraph[5], 3]

Out[162]= {{1 ↔ 2, 2 ↔ 5, 5 ↔ 4, 4 ↔ 3, 3 ↔ 5, 5 ↔ 1, 1 ↔ 4,
             4 ↔ 2, 2 ↔ 3, 3 ↔ 1}, {1 ↔ 2, 2 ↔ 5, 5 ↔ 4, 4 ↔ 3,
             3 ↔ 5, 5 ↔ 1, 1 ↔ 3, 3 ↔ 2, 2 ↔ 4, 4 ↔ 1}, {1 ↔ 2, 2 ↔ 5,
             5 ↔ 4, 4 ↔ 3, 3 ↔ 2, 2 ↔ 4, 4 ↔ 1, 1 ↔ 5, 5 ↔ 3, 3 ↔ 1}}
```

With the symbol `All` as the third argument, *Mathematica* will determine all of the Euler circuits. We see below that the complete graph on 5 vertices has 132 Euler circuits.

```
In[163]:= Length[FindEulerianCycle[CompleteGraph[5], All]]

Out[163]= 132
```

Now we'll have *Mathematica* help us to visualize this path by creating an animation that successively highlights the edges in the path. To do this we will use the `Animate` function. The `Animate` function takes two arguments, similar to `Table`. The first argument is a *Mathematica* expression, typically one that generates an image and which is dependent on a variable. The second argument is a list describing the range of the variable. The structure of this list is similar to the second argument in a `Table`.

We will also be making use of two options. Setting the `AnimationRunning` option to `False` will prevent the animation from beginning until you explicitly click on the play button. Without this option, the animation would run as soon as *Mathematica* has finished generating it. Setting the `AnimationRepetitions` option to 1 will cause the animation to stop once it has played through. This option defaults to `Infinity`, meaning it will automatically restart every time it reaches the end.

To create the animation, we need a graph and a circuit. We will use K_5 as the graph and we store the circuit as **`exampleCircuit`**. Note that we apply `First` since `FindEulerianCycle` returns a list of circuits and we want to access the first element of that list.

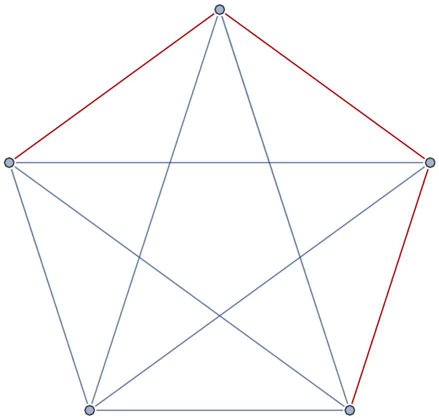
```
In[164]:= exampleCircuit = First[FindEulerianCycle[CompleteGraph[5]]]

Out[164]= {1 ⧟ 5, 5 ⧟ 4, 4 ⧟ 3, 3 ⧟ 5,
           5 ⧟ 2, 2 ⧟ 4, 4 ⧟ 1, 1 ⧟ 3, 3 ⧟ 2, 2 ⧟ 1}
```

To display the path, we will apply the HighlightGraph function. This function was first described in Section 10.2 in the subsection on bipartite graphs. It takes two arguments: a graph and a list of either vertices or edges.

To draw the successive stages in the circuit, we will apply HighlightGraph to the graph and to a sublist of **exampleCircuit**. We will obtain the sublist by applying Part (`[[…]]`) to a Span (`;;`) from 1 to the current stage. For example, to display the path after three steps, we enter the following.

```
In[165]:= HighlightGraph[CompleteGraph[5], exampleCircuit[[1 ;; 3]]]
```

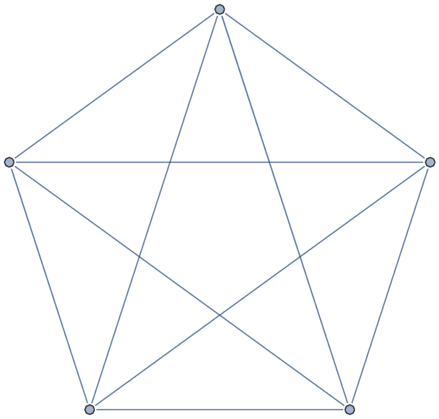
Out[165]=



Also note that the span from 1 to 0 will result in the empty list and thus nothing highlighted.

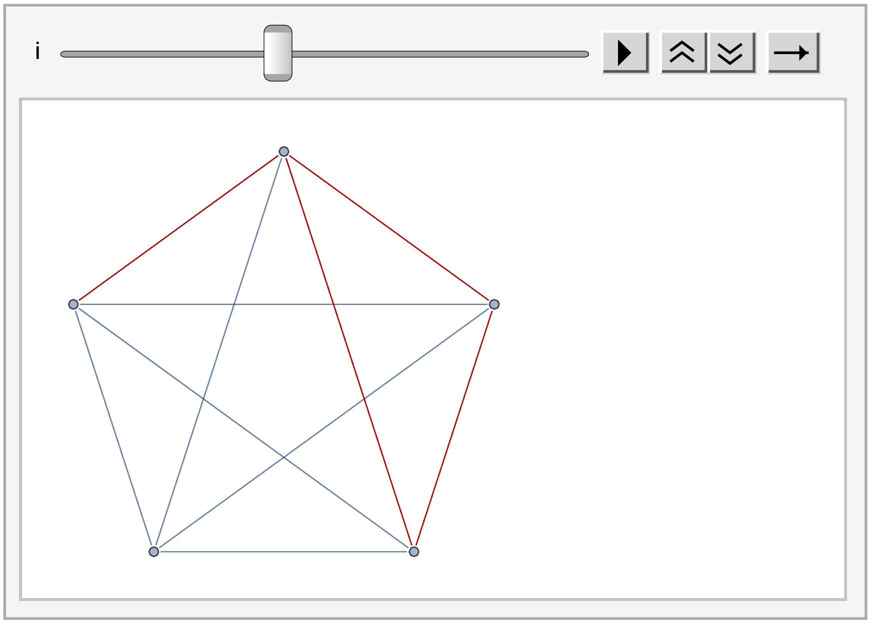
```
In[166]:= exampleCircuit[[1 ;; 0]]

Out[166]= {}

In[167]:= HighlightGraph[CompleteGraph[5], exampleCircuit[[1 ;; 0]]]
```

Out[167]=



We produce the animation using HighlightGraph as above, with the second argument to the Span (`;;`) as a variable. In the second argument of Animate, this variable will be set to run from 0 to the Length of the path. Note that, unlike Table, the variables in an Animate are not assumed to be restricted to integers, so we must specify a step value of 1 in the variable specification.

```
In[168]:= Animate[HighlightGraph[CompleteGraph[5],
            exampleCircuit[[1 ;; i]]], {i, 0, Length[exampleCircuit], 1},
           AnimationRunning → False, AnimationRepetitions → 1]
```

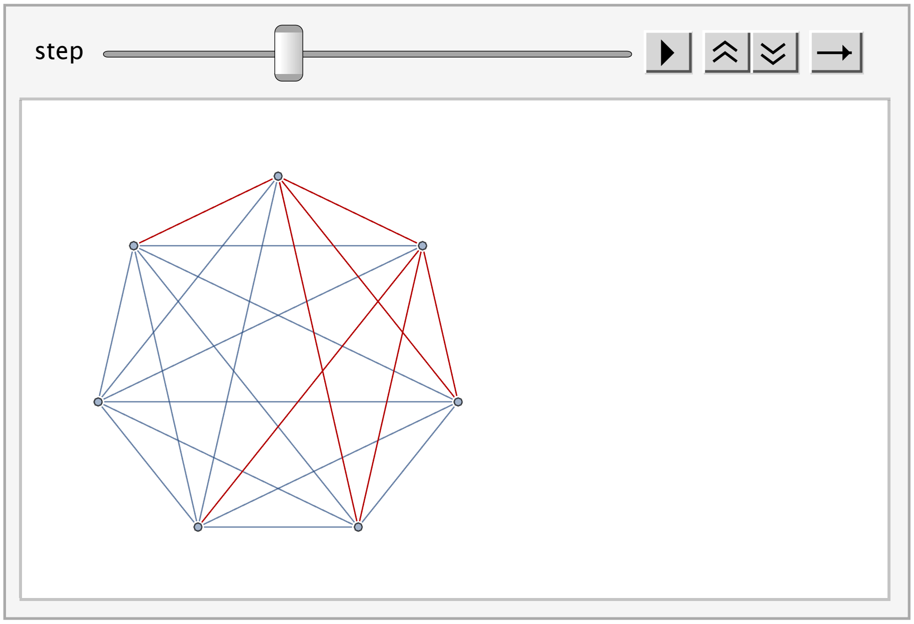


We now turn this into a function. The only difference between the **`animatePath`** function below and the example from above is that we replace the variable in the loop specification with the list **`{i,0,"step"}`**. This syntax gives the variable **`i`** the initial value 0 and labels it as “step” in the animation controller. Also note that we use a `BlankNullSequence` (`___`) to allow the **`animatePath`** function to take 0 or more arguments after the required graph and path arguments. We use this to pass options to the `HighlightGraph` function.

```
In[169]:= animatePath[g_Graph, p_List, opts___] := Module[{i, len},
            len = Length[p];
            Animate[HighlightGraph[g, p[[1 ;; i]], opts],
             {{i, 0, "step"}, 0, len, 1},
             AnimationRunning → False, AnimationRepetitions → 1]
           ]
```

To use this function, we just pass it an Eulerian graph, a circuit, and any options for drawing the graph.

```
In[170]:= animatePath[CompleteGraph[7],
            First[FindEulerianCycle[CompleteGraph[7]]],
            VertexLabels → "Name", ImagePadding → 10]
```



You can see the Euler circuit traced out by clicking on the play button.

Note that while our examples have all been undirected, the functions described here also apply to directed graphs.

Euler Circuits in Multigraphs

As usual, *Mathematica*'s built-in function does not apply to pseudographs. We will examine the problem of finding Euler circuits in undirected multigraphs. We know, from Theorem 1 of Section 10.5, that a connected multigraph with at least two vertices has an Euler circuit if and only if the degree of every vertex is even. It is easy to see that Theorem 1 extends to pseudographs as well. Using this fact, we can write a simple function for determining whether or not an undirected pseudograph has an Euler circuit. Note that, as we have done before, we input the undirected pseudograph as a list of rules, but the function assumes that the graph being modeled is undirected.

```
In[171]:= eulerianPseudographQ[edgeList : {___Rule}] := Module[{v, G},
             G = Graph[DeleteDuplicates[Sort /@ edgeList],
               DirectedEdges → False];
             Catch[
              If[! ConnectedGraphQ[G] || Length[VertexList[G]] < 2,
               Throw[False]];
              Do[If[OddQ[undirectedDegree[edgeList, v]], Throw[False]]
               , {v, VertexList[G]}];
              Throw[True]
             ]
            ]
```

We begin the **eulerianPseudographQ** function by forming an undirected `Graph` object. The purpose of this is to allow us to use some built-in functions rather than creating them from scratch. The `Graph` object is obtained from the input pseudograph as follows. We `Map` (**/@**) the `Sort` function over the **edgeList** of the input graph. The `Map` (**/@**) causes the `Sort` function to be applied to the elements of **edgeList**, rather than the list itself. That is, `Sort` will be applied to the individual rules that describe the edges of the graph, which causes the `Rule` (**->**) to point from lesser to greater, as illustrated below. This forces a canonical representation of each edge. Since the graph is undirected, this allows *Mathematica* to recognize 1 → 2 as the same as 2 → 1.

```
In[172]:= Sort[2 → 1]

Out[172]= 1 → 2
```

After the edges are put in canonical form with **Sort /@ edgeList**, we apply `DeleteDuplicates`. This results in the edge set of a simple graph but such that every pair of vertices that were connected in the pseudograph are still connected. We then use the resulting list as the argument to `Graph`, with the `DirectedEdges` option set to `False`. Note that if we had not mapped `Sort` onto the original edge list, we would risk an error. For example, if the input had included both 1 → 2 and 2 → 1, when *Mathematica* applied `Graph` with `DirectedEdges` false, it would then identify those as the same edge and identify the graph as having multiple edges. For a directed version of this function, `DirectedEdges` would be true, and you would not map `Sort` over the edge list.

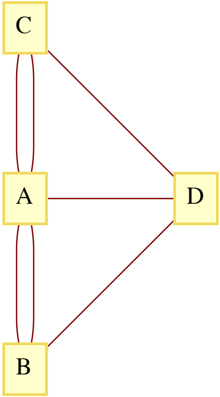
All of the tests are contained in `Catch`. If any of the conditions fail, they will `Throw` `False`. Otherwise, if the graph passes all of the hurdles, `True` will be thrown at the end of the block. The first test is to ensure that the graph is connected and that it has at least 2 vertices, which are conditions required for Theorem 1 to apply. This is where the creation of the `Graph` object is useful, as it allows us to apply the built-in functions `ConnectedGraphQ` and `VertexList` rather than create such functions for pseudographs. The second `If` is contained within a `Do` loop to test the degree of each vertex. We apply the **undirectedDegree** function, written in Section 10.2, and the Boolean function `OddQ`.

We can use this function to solve the Bridges of Königsberg problem. First we create a representation of the town and its bridges as a graph (this replicates Figure 2 in Section 10.5). Then we apply the test.

```
In[173]:= konigsbergEdges = {"A" → "B", "A" → "B",
             "A" → "C", "A" → "C", "A" → "D", "B" → "D", "C" → "D"}

Out[173]= {A → B, A → B, A → C, A → C, A → D, B → D, C → D}
```

In[174]:=
```
GraphPlot[konigsbergEdges,
 VertexLabeling → True, VertexCoordinateRules →
  {"A" → {0, 1}, "B" → {0, 0}, "C" → {0, 2}, "D" → {1, 1}}]
```

Out[174]=



In[175]:=
```
eulerianPseudographQ[konigsbergEdges]
```

Out[175]=
```
False
```

Now that we have a test that tells us if a circuit exists, we will implement Algorithm 1 from Section 10.5 in order to find an Euler circuit, if it exists. The following algorithm will find an Euler circuit in a multigraph. It could also be applied to a pseudograph without generating an error, but it will not include loops in the circuit.

```
In[176]:= findEulerianCycleMultigraph[edgeList : {___Rule}] :=
           Module[{H, circuit, subC, i, n,
             v, insertPoint, w, buildingSub, oldC},
            If[! eulerianPseudographQ[edgeList], Return[$Failed]];
            circuit = {};
            H = edgeList;
            While[H ≠ {},
             (* find a starting point *)
             If[circuit == {},
              subC = {H[[1]]};
              H = Delete[H, 1];
              insertPoint = 0,
              For[i = 1, i ≤ Length[circuit], i++,
               v = circuit[[i, 2]];
               n = neighbors[H, v];
               If[n ≠ {},
                w = n[[1]];
                subC = {v → w};
                insertPoint = i;
                H = DeleteCases[H, Rule[v, w] | Rule[w, v], {1}, 1];
                Break[]
               ]
              ]
             ];
             (* build a subcircuit *)
             buildingSub = True;
             While[buildingSub && H ≠ {},
              v = subC[[-1, 2]];
              w = First[neighbors[H, v]];
              H = DeleteCases[H, Rule[v, w] | Rule[w, v], {1}, 1];
              AppendTo[subC, v → w];
              If[w == subC[[1, 1]], buildingSub = False]
             ];
             (* splice the subcircuit into the main circuit *)
             If[circuit == {},
              circuit = subC,
              circuit = Flatten[Insert[circuit, subC, insertPoint + 1]]
             ]
            ];
            circuit
           ]
```

The function begins with a use of **eulerianPseudographQ** in order to avoid searching for a circuit that cannot exist. It then assigns to the symbol **H** a copy of the graph. It is this copy that is used throughout the rest of the function, rather than the input that was passed to the algorithm. The benefit of using a copy is that the function will be able to manipulate it as the algorithm proceeds, e.g., by deleting edges of `H` once they are included in the circuit so that those edges are not reused.

Recall the description of Algorithm 1 in Section 10.5. There are two key ideas at the heart of this algorithm. The first is that, for a graph whose vertices all have even degree, if you pick any vertex to start at and follow edges at random but without repetition, you will definitely return to the original vertex and create a circuit. The second key idea is that (for a connected graph), if your circuit does not include all of the edges of the graph, then some vertex used in the existing circuit can be made the starting point for a new subcircuit. This subcircuit can then be spliced into the main circuit. This will eventually use all the edges and the result will be a Euler circuit.

The symbol `circuit` will hold the main circuit that, at the end of the function, is output to the user. The circuit will be stored as a list of the edges through which the circuit passes and is initialized to the empty list. The main `While` loop consists of three parts: (1) determining the starting point for the subcircuit (named `subC`); (2) building the subcircuit; and (3) splicing the subcircuit into the main circuit.

The first step, finding the starting point for the subcircuit, depends on the state of the main circuit. If `circuit` is the empty list (i.e., this is the first pass through the main loop), then the starting point is the first edge in the graph. If the main circuit is not empty, then the else clause looks at the vertices in the main circuit to find one that has neighbors (since edges are deleted from `H` as they are added to the circuit, only vertices that are an endpoint of an unused edge have neighbors). The first vertex that has a neighbor is used as the starting point for the subcircuit. The `insertPoint` variable is used to keep track of the index, relative to `circuit`, of the starting vertex for the subcircuit. This is used when the subcircuit is spliced into the main circuit.

The second step is to build `subC`. The `buildingSub` symbol is used to control the `While` loop. It is initialized to true and is set to false once `subC` has returned to its starting vertex and is thus a circuit. The variable `v` is set to the last vertex currently included as part of the subcircuit and `w` represents a neighbor of `v`. To remove the edge between `v` and `w` from `H`, we use `DeleteCases`. The first argument is the list to delete from. The second argument is a pattern, in this case using `Alternatives` (`|`) to allow for either order of the vertices in the edge being deleted. If this function were to apply to a directed graph, the order would be important, but here we need to allow for the fact that the circuit may follow edges in either direction. The third argument is an optional level specification, with `{1}` indicating that only the first level is to be matched, that is, only members of the list and not substructures are considered. The final argument, which is also optional, places a limit on the number of matches to delete. With `1` in the final position, the function will only delete the first edge it finds, rather than all of them. Note that `{1}` is the default level specification for this function, but it must be included in order to use the fourth argument.

After deleting the edge from `H`, the newest vertex is compared with the starting vertex to determine if the circuit has been closed. If the new vertex closes the circuit, then the `buildingSub` variable is set to false, which causes the loop to terminate. Otherwise, the `While` loop continues building the subcircuit.

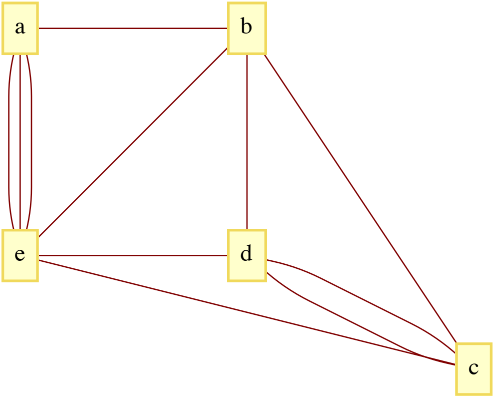
The third step, once the subcircuit has been built, is to splice it into the main circuit. If `circuit` is empty, it is merely set to `subC`. Otherwise, `Insert` is used to add the subcircuit. `Insert` takes a list as the first argument and adds the expression given as the second argument in the position specified by the third argument. The elements previously in that position and later and pushed down to make room.

We give the third argument as **`insertPoint + 1`** to insert the sublist between the edges at position **`insertPoint`** and position **`insertPoint + 1`**.

The main `While` loop continues until all the edges of the graph have been included in the circuit, making circuit an Euler circuit for the graph. As an example, consider Exercise 5 from Section 10.5.

```
In[177]:= exercise5Edges =
            {"a" → "b", "a" → "e", "a" → "e", "a" → "e", "b" → "c", "b" → "d",
             "b" → "e", "c" → "d", "c" → "d", "c" → "e", "d" → "e"}

Out[177]= {a → b, a → e, a → e, a → e, b → c,
           b → d, b → e, c → d, c → d, c → e, d → e}

In[178]:= GraphPlot[exercise5Edges, VertexLabeling → True,
           VertexCoordinateRules → {"a" → {0, 2}, "b" → {1, 2},
             "c" → {2, .5}, "d" → {1, 1}, "e" → {0, 1}}]
```

Out[178]=



```
In[179]:= exercise5EulerPath =
           findEulerianCycleMultigraph[exercise5Edges]

Out[179]= {a → b, b → c, c → d, d → e, e → a,
           a → e, e → c, c → d, d → b, b → e, e → a}
```

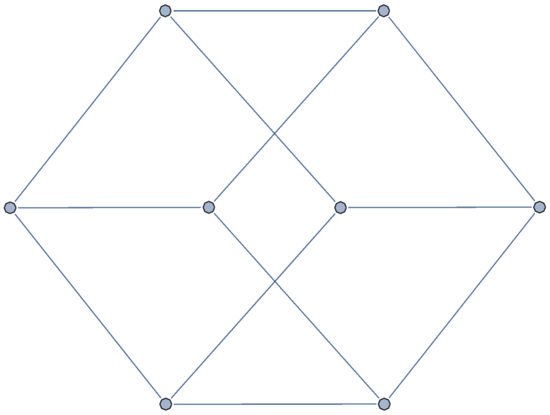
Note that the edge between *a* and *e* is traversed three times: twice in about the middle of the circuit immediately in succession, and once as the last edge in the cycle. This is consistent with there being three edges between *a* and *e*.

Hamilton Circuits

Turning our attention to Hamilton circuits, *Mathematica* provides the function `HamiltonianGraphQ` for determining whether or not the graph contains a Hamilton circuit. This command, like `EulerianGraphQ`, accepts a graph as the sole argument. It returns true or false depending on whether the graph has a Hamilton circuit.

```
In[180]:= hcGraphExample = HypercubeGraph[3]
```

Out[180]=



```
In[181]:= HamiltonianGraphQ[hcGraphExample]

Out[181]= True
```

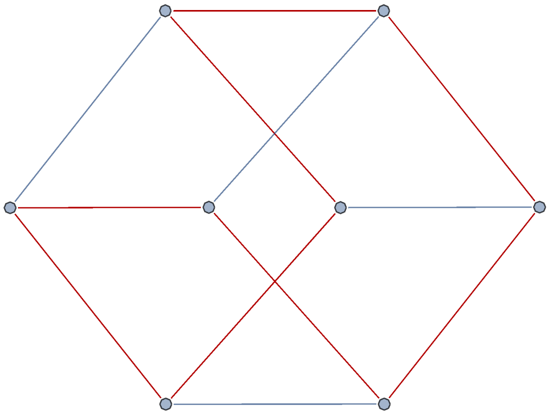
The `FindHamiltonianCycle` function, similar to `FindEulerianCycle`, accepts a graph as the argument and returns a list containing a list containing a Hamiltonian circuit. The optional second argument can be used to find more than one cycle.

```
In[182]:= FindHamiltonianCycle[hcGraphExample]

Out[182]= {{1 ⟷ 2, 2 ⟷ 4, 4 ⟷ 8, 8 ⟷ 6, 6 ⟷ 5, 5 ⟷ 7, 7 ⟷ 3, 3 ⟷ 1}}

In[183]:= FindHamiltonianCycle[hcGraphExample, All]

Out[183]= {{1 ⟷ 3, 3 ⟷ 7, 7 ⟷ 8, 8 ⟷ 4, 4 ⟷ 2, 2 ⟷ 6, 6 ⟷ 5, 5 ⟷ 1},
            {1 ⟷ 3, 3 ⟷ 4, 4 ⟷ 2, 2 ⟷ 6, 6 ⟷ 8, 8 ⟷ 7, 7 ⟷ 5, 5 ⟷ 1},
            {1 ⟷ 2, 2 ⟷ 6, 6 ⟷ 8, 8 ⟷ 4, 4 ⟷ 3, 3 ⟷ 7, 7 ⟷ 5, 5 ⟷ 1},
            {1 ⟷ 2, 2 ⟷ 4, 4 ⟷ 3, 3 ⟷ 7, 7 ⟷ 8, 8 ⟷ 6, 6 ⟷ 5, 5 ⟷ 1},
            {1 ⟷ 2, 2 ⟷ 6, 6 ⟷ 5, 5 ⟷ 7, 7 ⟷ 8, 8 ⟷ 4, 4 ⟷ 3, 3 ⟷ 1},
            {1 ⟷ 2, 2 ⟷ 4, 4 ⟷ 8, 8 ⟷ 6, 6 ⟷ 5, 5 ⟷ 7, 7 ⟷ 3, 3 ⟷ 1}}

In[184]:= hcGraphExampleCircuit =
           First[FindHamiltonianCycle[hcGraphExample]]

Out[184]= {1 ⟷ 2, 2 ⟷ 4, 4 ⟷ 8, 8 ⟷ 6, 6 ⟷ 5, 5 ⟷ 7, 7 ⟷ 3, 3 ⟷ 1}
```

We can use `HighlightGraph` to illustrate the path statically.

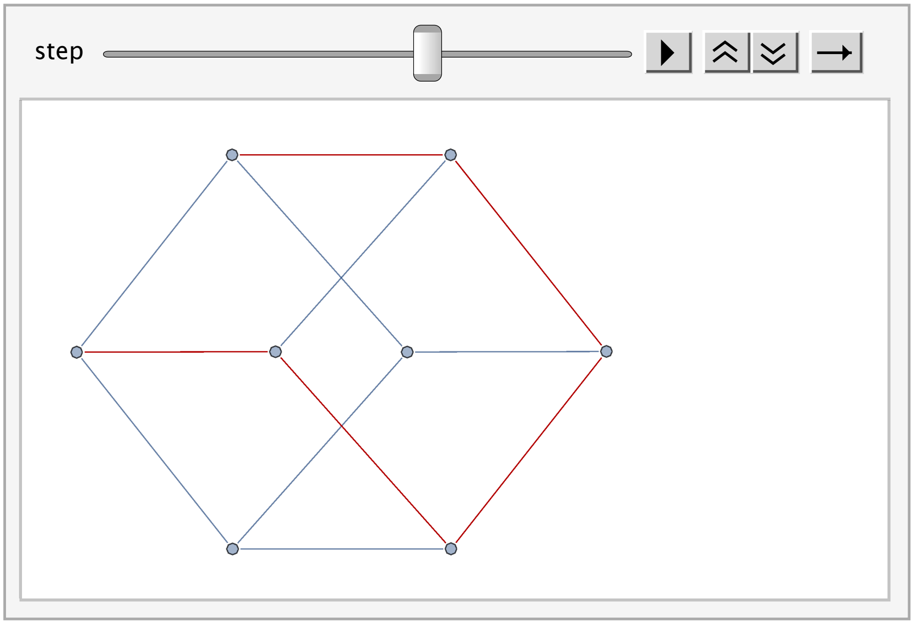
In[185]:= **HighlightGraph[hcGraphExample, hcGraphExampleCircuit]**

Out[185]=



Also, our animation function **animatePath**, created above, works equally well here.

In[186]:= **animatePath[hcGraphExample, hcGraphExampleCircuit]**



Note that a pseudograph is Hamiltonian if and only if its underlying simple graph is Hamiltonian, so there is no need for us to extend the built-in functions to pseudographs.

10.6 Shortest-Path Problems

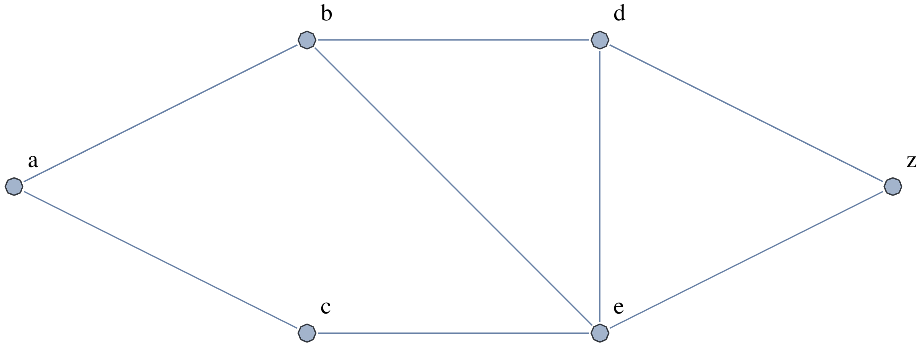
Among the most common problems in graph theory are shortest path problems. Generally, in shortest path problems, we wish to determine a path between two vertices of a weighted graph that is minimal in terms of the total weight of the edges in the path.

To define a `Graph` object with weighted edges, you use the `EdgeWeight` option. The value associated with the option is the list of the weights of each edge. The weights must appear in the same order as they are displayed in the output by `EdgeList`. For graphs you define by listing the edges, this is identical to the order you give in the definition.

We reproduce Exercise 2 from Section 10.6 of the textbook to use as an example.

```
In[187]:= exercise2 = Graph[{"a", "b", "c", "d", "e", "z"},
            {"a" → "b", "a" → "c", "b" → "d", "b" → "e", "c" → "e",
             "d" → "e", "d" → "z", "e" → "z"}, DirectedEdges → False,
            EdgeWeight -> {2, 3, 5, 2, 5, 1, 2, 4}, VertexCoordinates →
             {{0, .5}, {1, 1}, {1, 0}, {2, 1}, {2, 0}, {3, .5}},
            VertexLabels → "Name", ImagePadding → 5]
```

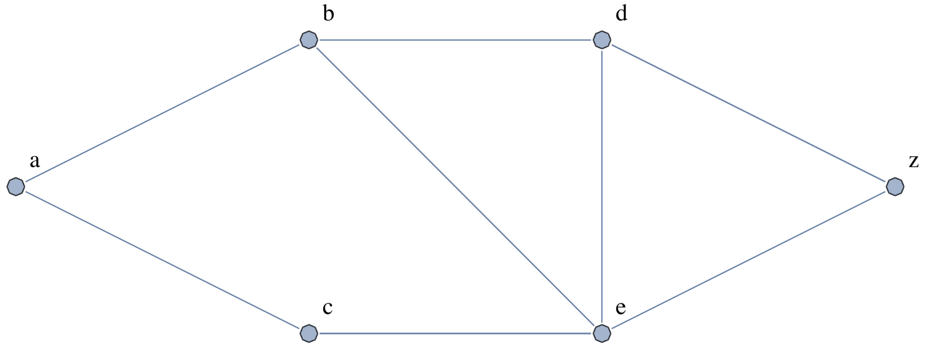
Out[187]=



A second approach to setting edge weights is by using the `EdgeWeight` property. Rather than listing all of the edges and then listing the weights separately using the option as shown above, we can wrap the edge definitions in the `Property` wrapper, setting the edge weights at the same time the edges are described.

```
In[188]:= Graph[{"a", "b", "c", "d", "e", "z"},
            {Property["a" → "b", EdgeWeight → 2],
             Property["a" → "c", EdgeWeight → 3],
             Property["b" → "d", EdgeWeight → 5],
             Property["b" → "e", EdgeWeight → 2],
             Property["c" → "e", EdgeWeight → 5],
             Property["d" → "e", EdgeWeight → 1],
             Property["d" → "z", EdgeWeight → 2],
             Property["e" → "z", EdgeWeight → 4]},
           DirectedEdges → False, VertexCoordinates →
             {{0, .5}, {1, 1}, {1, 0}, {2, 1}, {2, 0}, {3, .5}},
           VertexLabels → "Name", ImagePadding → 5]
```

Out[188]=



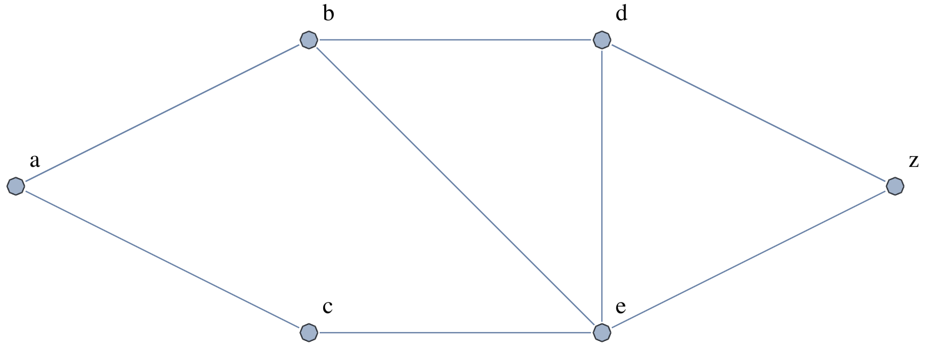
You can also define a weighted graph using the `WeightedAdjacencyGraph` function and an adjacency matrix. Unlike with `AdjacencyGraph`, the adjacency matrix must use `Infinity`, or ∞ (ESCinfESC) to indicate that there is no edge between the corresponding vertices. In the example below, we give the list of vertices as the first argument, otherwise *Mathematica* will automatically use positive integers to name the vertices. Note that `WeightedAdjacencyMatrix` applied to a weighted graph returns the adjacency matrix with weights.

```
In[189]:= exercise2matrix =
             {{∞, 2, 3, ∞, ∞, ∞}, {2, ∞, ∞, 5, 2, ∞}, {3, ∞, ∞, ∞, 5, ∞},
              {∞, 5, ∞, ∞, 1, 2}, {∞, 2, 5, 1, ∞, 4}, {∞, ∞, ∞, 2, 4, ∞}};
          exercise2matrix // MatrixForm
```

```
Out[190]//MatrixForm=
          ∞  2  3  ∞  ∞  ∞
          2  ∞  ∞  5  2  ∞
          3  ∞  ∞  ∞  5  ∞
          ∞  5  ∞  ∞  1  2
          ∞  2  5  1  ∞  4
          ∞  ∞  ∞  2  4  ∞
```

```
In[191]:= WeightedAdjacencyGraph[{"a", "b", "c", "d", "e", "z"},
           exercise2matrix, VertexCoordinates →
            {{0, .5}, {1, 1}, {1, 0}, {2, 1}, {2, 0}, {3, .5}},
           VertexLabels → "Name", ImagePadding → 5]
```

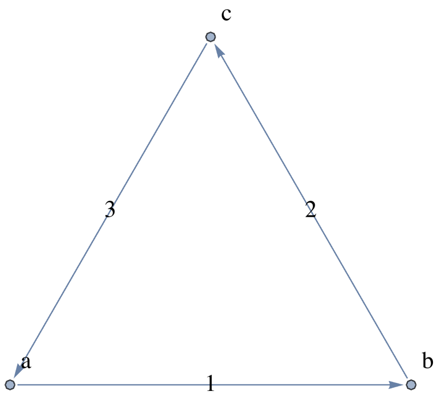
Out[191]=



Observe that the edge weights are not automatically displayed. We can display them with `EdgeLabels`, but unlike `VertexLabels`, there is no simple value, like **"Name"**, that will cause the weights to appear. Rather, we must set each edge's label individually. This is illustrated below.

```
In[192]:= Graph[{"a", "b", "c"},
           {"a" → "b", "b" → "c", "c" → "a"}, EdgeWeight → {1, 2, 3},
           VertexLabels → "Name", ImagePadding → 10,
           EdgeLabels → {"a" ⟼ "b" → 1, "b" ⟼ "c" → 2, "c" ⟼ "a" → 3}]
```

Out[192]=



Note that, even in a directed graph, rules cannot be used to describe edges within the `EdgeLabels` value. Rather the symbols ⟼ (⎋de⎋) or ⊶ (⎋ue⎋) must be used.

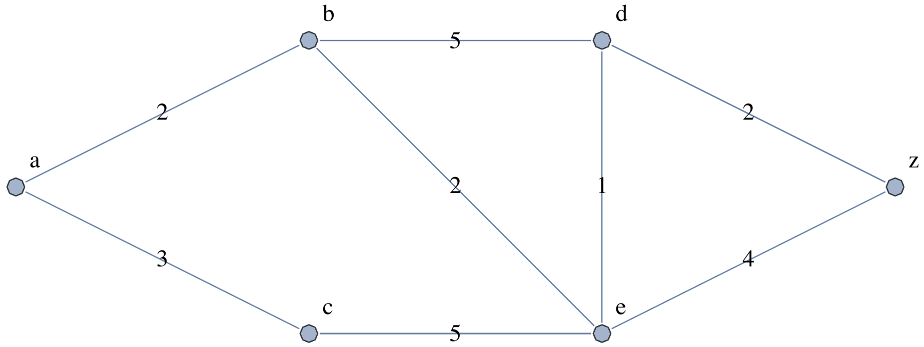
For graphs of any size, needing to assign the edge weights for the `EdgeWeight` option can be cumbersome. For convenience, we create the following function, which modifies a given graph by setting the `EdgeWeight` property to be the edge weight.

```
In[193]:= addWeightLabels[G_Graph] := Module[{E, W, H, i},
             H = G;
             Do[
              PropertyValue[{H, e}, EdgeLabels] =
               PropertyValue[{H, e}, EdgeWeight]
              , {e, EdgeList[H]}];
             H
            ]
```

This can be applied to an existing graph to display the labels once.

```
In[194]:= addWeightLabels[exercise2]
```

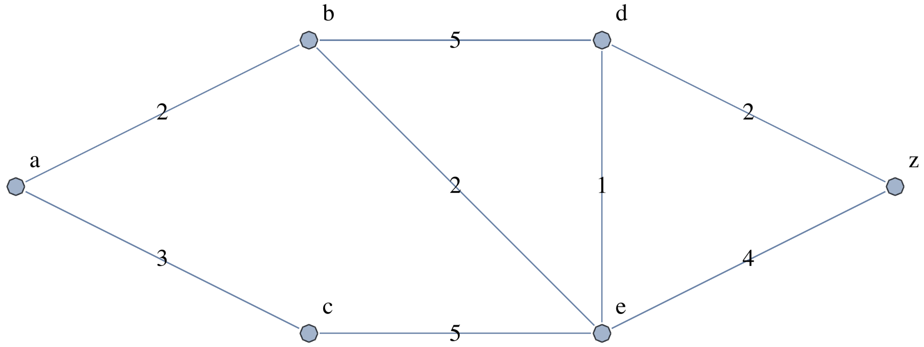
Out[194]=



Or it can be applied at the time the graph is created to make the labels permanent.

```
In[195]:= exercise2 = Graph[{"a", "b", "c", "d", "e", "z"},
              {"a" → "b", "a" → "c", "b" → "d", "b" → "e", "c" → "e",
               "d" → "e", "d" → "z", "e" → "z"}, DirectedEdges → False,
              EdgeWeight -> {2, 3, 5, 2, 5, 1, 2, 4}, VertexCoordinates →
               {{0, .5}, {1, 1}, {1, 0}, {2, 1}, {2, 0}, {3, .5}},
              VertexLabels → "Name", ImagePadding → 5] // addWeightLabels
```

Out[195]=



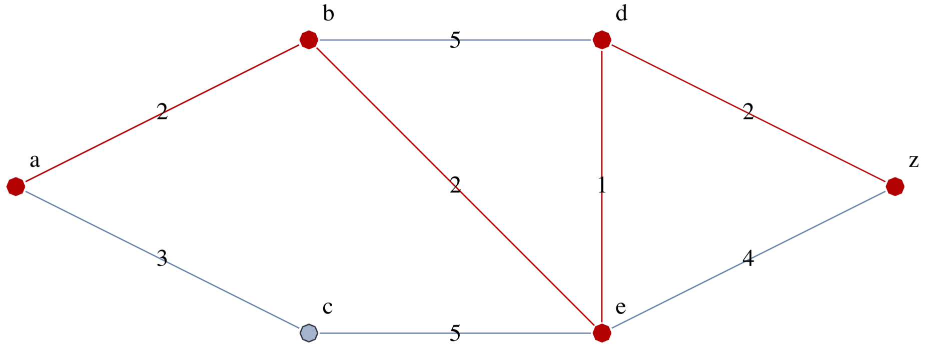
Now we will make use of *Mathematica*'s implementation of Dijkstra's algorithm to compute the shortest path between *a* and *z*. To do this, we simply call the `FindShortestPath` function with three arguments: the graph and the names of the starting and ending vertices. The output will be a list of vertices beginning with the starting vertex and ending with the final vertex through which the path runs.

```
In[196]:= FindShortestPath[exercise2, "a", "z"]

Out[196]= {a, b, e, d, z}
```

The `PathGraph` function can be use to help visualize this path. `PathGraph` accepts a list of vertices as input and produces the graph obtained by connecting successive vertices with an edge. By default, the graph produced is undirected. To obtain a directed path, assign the `DirectedEdges` option to `True`. Here, we will display the path in the graph by applying `HighlightGraph`, using `PathGraph` in the second argument.

```
In[197]:= HighlightGraph[exercise2,
           PathGraph@FindShortestPath[exercise2, "a", "z"]]
```

Out[197]=



The length of the shortest path can be determined with the `GraphDistance` function and the same arguments.

```
In[198]:= GraphDistance[exercise2, "a", "z"]

Out[198]= 7.
```

If the final argument, the destination vertex, is omitted from `GraphDistance`, the result will be a list of the lengths of the shortest paths to each vertex of the graph.

```
In[199]:= GraphDistance[exercise2, "a"]

Out[199]= {0, 2., 3., 5., 4., 7.}
```

To determine the shortest path from every vertex to every other vertex, use the `GraphDistanceMatrix` function. Algorithm 2 in the Exercises of Section 10.6 describes the Floyd-Warshall algorithm (also known as simply the Floyd algorithm), which is one method for computing this matrix.

```
In[200]:= GraphDistanceMatrix[exercise2] // MatrixForm

Out[200]//MatrixForm=
          0.  2.  3.  5.  4.  7.
          2.  0.  5.  3.  2.  5.
          3.  5.  0.  6.  5.  8.
          5.  3.  6.  0.  1.  2.
          4.  2.  5.  1.  0.  3.
          7.  5.  8.  2.  3.  0.
```

10.7 Planar Graphs

This section explains how *Mathematica* can be used to explore the question of whether a graph is planar by manipulating graphs in order to produce homeomorphic graphs and applying Kuratowski's Theorem. We consider only undirected simple graphs in this section. The question of planarity for a directed graph can be answered by considering the underlying undirected graph, which can be obtained by applying the function `UndirectedGraph`.

Before looking at how *Mathematica* can help you manipulate graphs to apply Kuratowski's Theorem manually, note that the function `PlanarGraphQ`, applied to a `Graph` object, will determine whether or not the graph is planar.

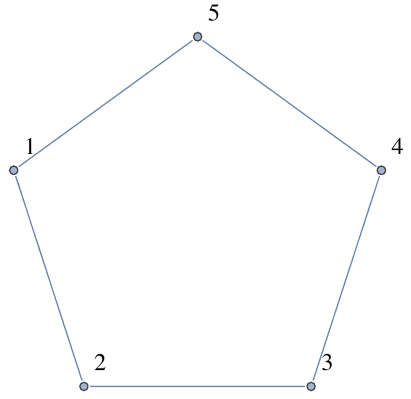
```
In[201]:= PlanarGraphQ[HypercubeGraph[3]]

Out[201]= True

In[202]:= PlanarGraphQ[CompleteGraph[5]]

Out[202]= False
```

Elementary Subdivisions, Smoothing, and Homeomorphic Graphs

Recall that an elementary subdivision refers to the process of modifying a graph by removing an edge $\{u, v\}$ and replacing it with a vertex w and new edges $\{u, w\}$ and $\{w, v\}$. Effectively, this splits the original edge into two by inserting a vertex in the middle of it. It is not difficult to use *Mathematica* to perform an elementary subdivision. We will use a cycle graph as an example.

```
In[203]:= subdivideExample =
           CycleGraph[5, VertexLabels → "Name", ImagePadding → 10]
```

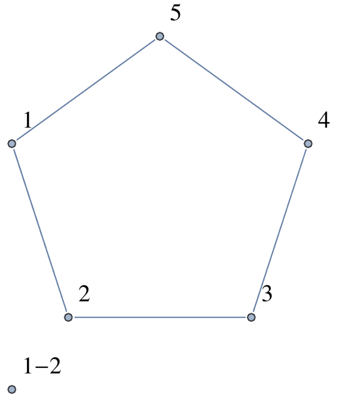
Out[203]=



We will subdivide the edge {1, 2}. To do this, we first need to introduce a vertex. We will give this vertex the name "1-2". We apply `ToString` to the integers to create strings and combine them and the hyphen with the `StringJoin` (**<>**) operator. We use the `VertexAdd` function to add the vertex to the graph.

```
In[204]:= subdivideExampleB =
           VertexAdd[subdivideExample, ToString[1] <> "-" <> ToString[2]]
```

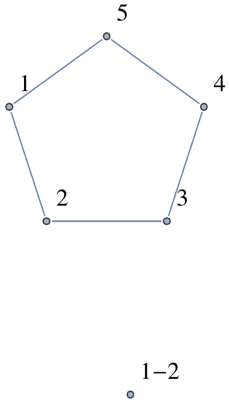
Out[204]=



Note that *Mathematica* has moved the original vertices. We can put them back in place by using `PropertyValue` to assign the `VertexCoordinates` property in the new graph to the position in the old.

```
In[205]:= Do[PropertyValue[{subdivideExampleB, v}, VertexCoordinates] =
             PropertyValue[{subdivideExample, v}, VertexCoordinates],
           {v, VertexList[subdivideExample]}]

In[206]:= subdivideExampleB
```

Out[206]=

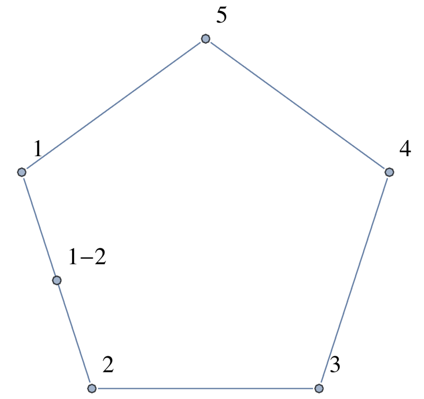


Now we place the new vertex along the edge which is to be deleted. This is done by using `PropertyValue` and averaging the positions of the existing vertices.

```
In[207]:= PropertyValue[{subdivideExampleB, "1-2"}, VertexCoordinates] =
           (PropertyValue[{subdivideExample, 1}, VertexCoordinates] +
              PropertyValue[{subdivideExample, 2}, VertexCoordinates]) / 2

Out[207]= {-0.769421, -0.25}
```

```
In[208]:= subdivideExampleB
```

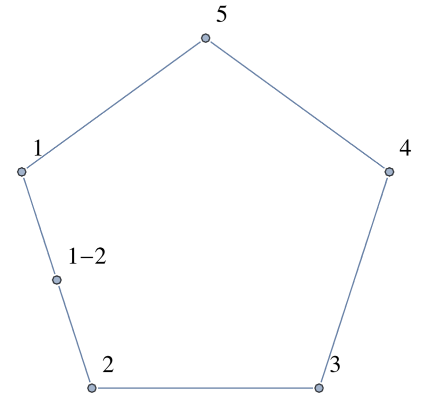
Out[208]=



Now we simply remove the original edge with `EdgeDelete` and add the new ones with `EdgeAdd`. Recall that the second argument of `EdgeDelete` must be given using the special symbols •–• (ESC ue ESC) or •⟶ (ESC de ESC), not as a `Rule` (**->**).

```
In[209]:= subdivideExampleB = EdgeDelete[subdivideExampleB, 1 •–• 2];
          subdivideExampleB = EdgeAdd[subdivideExampleB, 1 → "1-2"];
          subdivideExampleB = EdgeAdd[subdivideExampleB, 2 → "1-2"]
```

Out[211]=



Based on this example, we create the following function.

```
In[212]:= subdivideGraph::nonedge =
             "Second argument must be an edge in the graph.";
           subdivideGraph[G_Graph, E_Rule | E_UndirectedEdge] /;
             UndirectedGraphQ[G] := Module[{H, e, newV, v},
             If[Head[E] === Rule,
              e = UndirectedEdge @@ E,
              e = E
             ];
             If[! EdgeQ[G, e],
              Message[subdivideGraph::nonedge]; Return[$Failed]];
             newV = ToString[e[[1]]] <> "-" <> ToString[e[[2]]];
             H = VertexAdd[G, newV];
             Do[PropertyValue[{H, v}, VertexCoordinates] =
               PropertyValue[{G, v}, VertexCoordinates],
              {v, VertexList[G]}];
             PropertyValue[{H, newV}, VertexCoordinates] =
              (PropertyValue[{H, e[[1]]}, VertexCoordinates] +
                 PropertyValue[{H, e[[2]]}, VertexCoordinates]) / 2;
             H = EdgeDelete[H, e];
             H = EdgeAdd[H, e[[1]] → newV];
             H = EdgeAdd[H, newV → e[[2]]];
             H
            ]
```

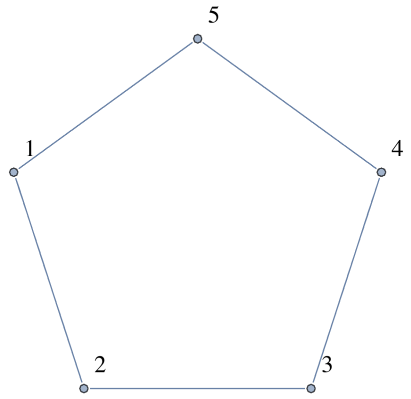
The inverse operation of elementary subdivision is referred to as smoothing. To be precise, let *v* be a vertex of degree 2 with neighbors *u* and *w* and such that *u* and *w* are not adjacent. We smooth the vertex *v* by deleting *v* and the edges incident to it and adding the edge {*u*, *w*}. Below we have created a function to implement smoothing.

```
In[214]:= smoothGraph::vertx = "Cannot smooth this vertex.";
           smoothGraph[G_Graph, v_] /; UndirectedGraphQ[G] :=
            Module[{N, H, e},
             N = AdjacencyList[G, v];
             If[Length[N] ≠ 2 || EdgeQ[G, UndirectedEdge[N[[1]], N[[2]]]],
              Message[smoothGraph::vertx]; Return[$Failed]];
             H = VertexDelete[G, v];
             e = UndirectedEdge[N[[1]], N[[2]]];
             H = EdgeAdd[H, e];
             H
            ]
```

As an example, we can smooth the vertex “1-2” added above.

In[216]:=
```
smoothGraph[subdivideExampleB, "1-2"]
```

Out[216]=



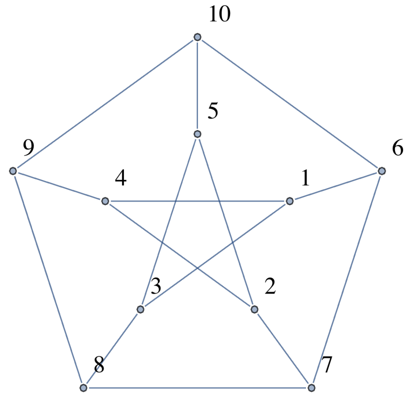
The textbook defines graphs to be homeomorphic if they can be obtained from the same graph from a sequence of elementary subdivisions. It is clear that if $G_1, G_2, G_3, \ldots, G_n$ is a sequence of graphs, each of which can be obtained from the previous by an elementary subdivision, then $G_n, \ldots, G_3, G_2, G_1$ is a sequence of graphs, each of which can be obtained from the previous by a smoothing. So we can say that two graphs are homeomorphic if one can be transformed into the other by a sequence of elementary subdivisions and smoothings.

Applying Kuratowski's Theorem

Recall that Kuratowski's Theorem asserts that a graph is nonplanar if and only if it contains a subgraph homeomorphic to either $K_{3,3}$ or K_5. Using the functions above and those for creating subgraphs, we can use *Mathematica* to manipulate a graph and confirm that it is nonplanar using Kuratowski's Theorem. We will illustrate this with the Petersen graph.

In[217]:=
```
petersen =
  PetersenGraph[5, 2, VertexLabels → "Name", ImagePadding → 10]
```

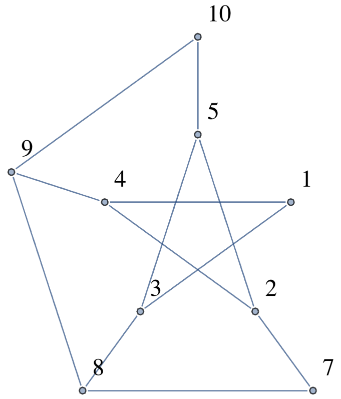
Out[217]=



First, we form the subgraph of the Petersen graph obtained by removing vertex 2 and the three edges incident to it.

In[218]:=
```
petersen1 = VertexDelete[petersen, 6]
```

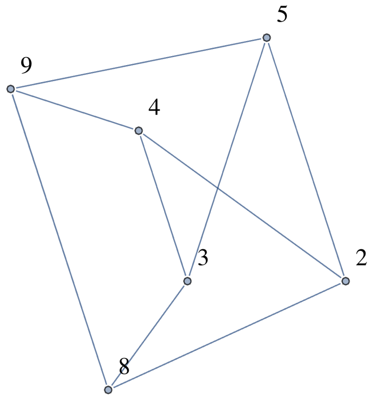
Out[218]=



Now we notice that there are three vertices that are smoothable: 1, 7, and 10. That is to say, those three vertices have degree 2 and their neighbors are not adjacent.

In[219]:=
```
petersen2 = smoothGraph[petersen1, 1];
petersen3 = smoothGraph[petersen2, 7];
petersen4 = smoothGraph[petersen3, 10]
```

Out[220]=



We now observe that this graph has 6 vertices, each of which has degree 3, just like $K_{3,3}$. So there is a definite possibility that this graph is $K_{3,3}$. We have *Mathematica* confirm that is $K_{3,3}$ with `IsomorphicGraphQ`.

In[221]:=
```
IsomorphicGraphQ[petersen4, CompleteGraph[{3, 3}]]
```

Out[221]=
```
True
```

This demonstrates that the Petersen graph has a subgraph that is homeomorphic to $K_{3,3}$ and hence is nonplanar.

10.8 Graph Coloring

In this section we consider the problem of how to properly color a graph; that is, how to assign to each vertex of a graph a color such that no vertex has the same color as any of its neighbors.

It is worth noting that, in terms of computational complexity, Hamilton circuits and graph coloring are equivalently difficult problems.

A Greedy Coloring Algorithm

We will create a function based on the algorithm described in the preface to Exercise 29 in Section 10.8 of the text. It can be shown that this algorithm will color a graph using at most one more color than the maximal degree of the graph. It is considered a greedy algorithm because it makes optimal choices at each step but never reconsiders its choices. That is to say, it does the best it can at every step but never backtracks to make improvements. Greedy algorithms often lead to good, but non-optimal, solutions.

The algorithm proceeds as follows. First, the vertices are sorted in order of descending degree. The first color is assigned to the first vertex in the list. Also assign color 1 to the first vertex in the list not adjacent to vertex 1, to the next vertex not adjacent to those already colored, etc. Then move on to the second color. The first uncolored vertex in the list is assigned color 2, as are vertices further down the list not adjacent to ones previously assigned the second color. This continues until all of the vertices have been given a color.

Our first step in implementing this function will be to sort the list of vertices in decreasing order of degree. For this, we will make use of *Mathematica*'s very flexible `Sort` function. With no additional instructions, *Mathematica* will sort a list of numbers in increasing numerical order and a list of strings in lexicographical order. But the `Sort` function takes an optional argument that allows us to specify the way in which the list is sorted. Specifically, `Sort` takes as an argument a Boolean-valued function on two arguments and returns true if the first argument precedes the second.

For our graph coloring procedure, we will create a helper function that takes a `Graph` object and returns the sorted list of vertices.

```
In[222]:= sortVertices[G_Graph] := Sort[VertexList[G],
            VertexDegree[G, #1] ≥ VertexDegree[G, #2] &]
```

In order for our algorithm to color the vertices of a graph, we need to decide on what colors to use. We define a list of colors globally.

```
In[223]:= colorList = {Red, Green, Blue, Magenta,
            Orange, Pink, Purple, Cyan, Brown, Black}

Out[223]= {RGBColor[1, 0, 0], RGBColor[0, 1, 0],
           RGBColor[0, 0, 1], RGBColor[1, 0, 1], RGBColor[1, 0.5, 0],
           RGBColor[1, 0.5, 0.5], RGBColor[0.5, 0, 0.5],
           RGBColor[0, 1, 1], RGBColor[0.6, 0.4, 0.2], GrayLevel[0]}
```

Now we will implement the greedy coloring algorithm.

```
In[224]:= greedyColorer::colorx = "Insufficiently many colors.";
          greedyColorer[G_Graph] :=
           Module[{H = G, V, currentColor, excludeSet, i},
            V = sortVertices[H];
            For[currentColor = 1,
             currentColor ≤ Length[colorList], currentColor++,
             PropertyValue[{H, V[[1]]}, VertexStyle] =
              colorList[[currentColor]];
             excludeSet = VertexList[NeighborhoodGraph[H, V[[1]]]];
             V = Delete[V, 1];
             i = 1;
             While[i ≤ Length[V],
              If[! MemberQ[excludeSet, V[[i]]],
               PropertyValue[{H, V[[i]]}, VertexStyle] =
                colorList[[currentColor]];
               excludeSet = Union[excludeSet,
                 VertexList[NeighborhoodGraph[H, V[[i]]]]];
               V = Delete[V, i],
               i++
              ]
             ];
             If[V == {}, Break[]]
            ];
            If[V ≠ {},
             Message[greedyColorer::colorx]; Return[$Failed],
             H]
           ]
```

Note that the list **V**, which is initialized to the list of vertices, sorted in decreasing order of degree, is used to track which vertices still need to be assigned a color. When a vertex has been assigned a color, it is deleted from the list **V** using `Delete`. The `Delete` function removes from the list in the first argument the element at the position specified by the second argument.

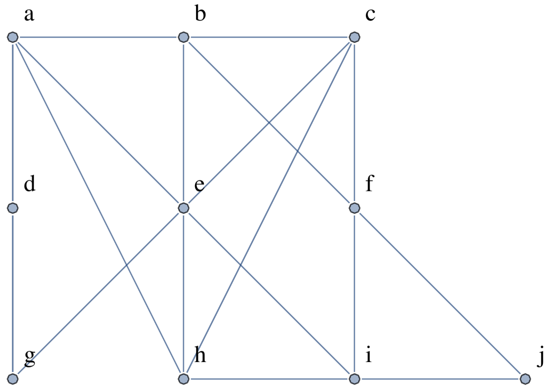
The **excludeSet** variable is used to store all vertices which cannot be assigned the current color. Each time a vertex is assigned a color, it and all of its neighbors are added to the **excludeSet**. As the function looks down the list of vertices that still need to have a color assigned, it checks to see if they are in this set.

The index **i**, which controls the `While` loop, is incremented in the else clause of the `If` statement that tests to see if a vertex can be assigned the color. If the vertex at index **i** is assigned the color, then it is removed from the list **V**, and thus the index **i** refers to a different vertex (the vertex previously in position $i + 1$).

As an example, we solve Exercise 29 of Section 10.8.

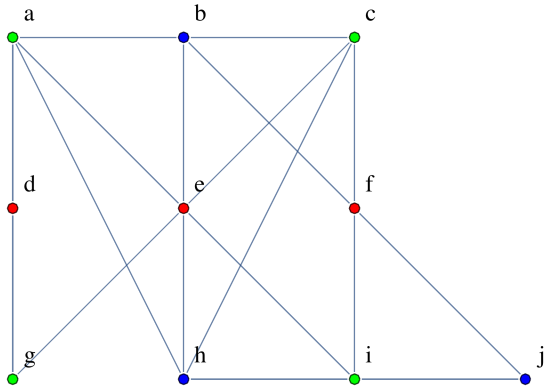
```
In[226]:= exercise29 =
           Graph[{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"},
            {"a" → "b", "a" → "d", "a" → "e", "a" → "h", "b" → "c",
             "b" → "e", "b" → "f", "c" → "e", "c" → "f", "c" → "h",
             "d" → "g", "e" → "g", "e" → "h", "e" → "i", "f" → "i",
             "f" → "j", "h" → "i", "i" → "j"}, DirectedEdges → False,
            VertexCoordinates → {{0, 2}, {1, 2}, {2, 2}, {0, 1},
              {1, 1}, {2, 1}, {0, 0}, {1, 0}, {2, 0}, {3, 0}},
            VertexLabels → "Name", ImagePadding → 10]
```

Out[226]=



```
In[227]:= greedyColorer[exercise29]
```

Out[227]=



Solutions to Computer Projects and Computations and Explorations

Computations and Explorations 1

Display all simple graphs with four vertices.

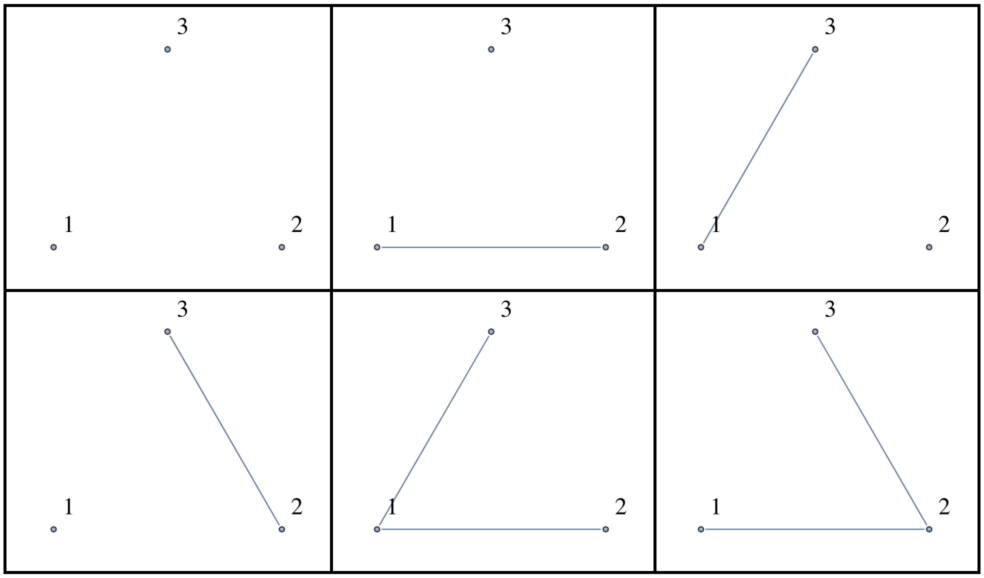
Solution: To solve this problem, we will generate all possible edge sets and then construct the graphs based on these edge sets. The possible edge sets are all of the subsets of the set of all possible edges, which we obtain from the complete graph on the vertices. We will generalize the question and have our function create all the simple graphs on n vertices.

```
In[228]:= allGraphs[n_Integer] /; n > 0 :=
            Module[{cg, v, A = {}, V = Range[n], powerE, vCoords, E},
             cg = CompleteGraph[n];
             powerE = Subsets[EdgeList[cg]];
             vCoords = Table[
               PropertyValue[{cg, v}, VertexCoordinates], {v, Range[n]}];
             Do[AppendTo[A, Graph[V, E, VertexLabels → "Name",
                ImagePadding → 10, VertexCoordinates → vCoords]]
              , {E, powerE}];
             A
            ]
```

Recall that the complete graph on n vertices has $C(n, 2)$ edges, so there are $2^{C(n,2)}$ graphs on n vertices. So on 4 vertices, there are 64 graphs. For $n = 3$, there are only 8 graphs, which is more manageable.

We use the Partition function to break the list of all graphs into triples and then apply Grid with the Frame option in order to display the results in a useful way.

```
In[229]:= Grid[Partition[allGraphs[3], 3], Frame → All]
```



Computations and Explorations 2

Display a full set of nonisomorphic simple graphs with six vertices.

Solution: The solution to this exercise is very similar to the previous question. The only difference is that, once the list of graphs is generated, we remove those that are isomorphic to others by applying

DeleteDuplicates with second argument IsomorphicGraphQ. This uses IsomorphicGraphQ to determine whether two elements are to be considered duplicates or not.

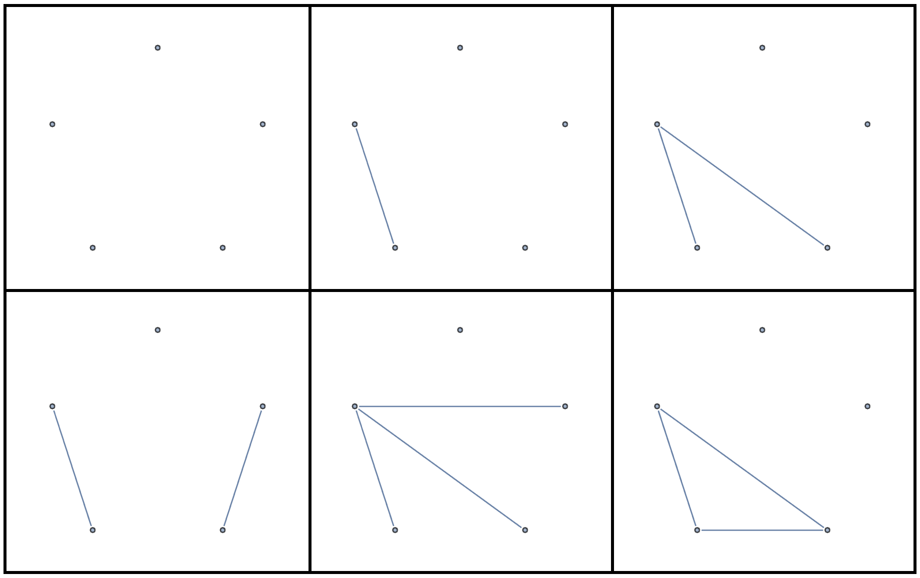
```
In[230]:= nonIsoGraphs[n_Integer] /; n > 0 := Module[
            {A = {}, cg, v, V = Range[n], powerE, vCoords, E, i, G, j},
            cg = CompleteGraph[n];
            powerE = Subsets[EdgeList[cg]];
            vCoords = Table[
              PropertyValue[{cg, v}, VertexCoordinates], {v, Range[n]}];
            Do[AppendTo[A, Graph[V, E, ImagePadding → 10,
                VertexCoordinates → vCoords]]
             , {E, powerE}];
            DeleteDuplicates[A, IsomorphicGraphQ]
           ]
```

We apply this to five vertices, since six takes a bit more time to compute.

```
In[231]:= nonIso5 = nonIsoGraphs[5];

In[232]:= Length[nonIso5]

Out[232]= 34
```

We see that there are 34 nonisomorphic graphs on 5 vertices. Here are the first eight.

```
In[233]:= Grid[Partition[nonIso5[[1 ;; 8]], 3], Frame → All]
```



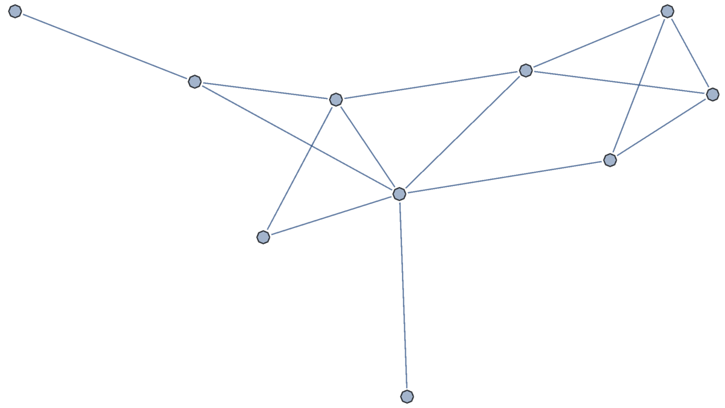
Computations and Explorations 9

> Generate at random simple graphs with 10 vertices. Stop when you have constructed one with an Euler circuit. Display an Euler circuit in this graph.

Solution: To generate the random graphs, we will use the RandomGraph function. By passing this function a list containing a number of vertices and a number of edges, it produces a graph with that number of vertices and edges. To display a random graph on 10 vertices and 15 edges, we enter the following.

```
In[234]:= RandomGraph[{10, 15}]
```

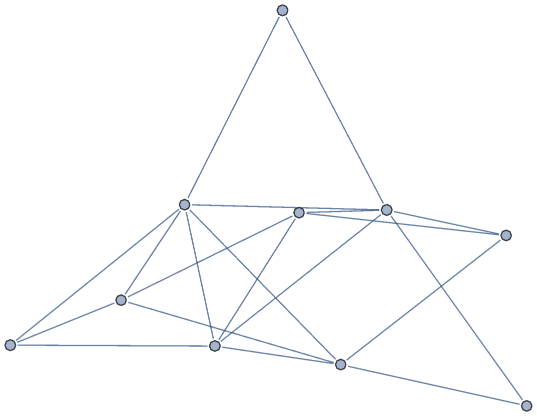
Out[234]=



Alternately, the argument to `RandomGraph` can be a graph distribution, which is a probability distribution on the sample space consisting of graphs. There are six built-in graph distributions. We will use the `BernoulliGraphDistribution`, which allows you to specify a number of vertices and a probability. The probability indicates the independent probability that an edge will appear between any two vertices. It accepts two arguments, the number of vertices and the probability. The following constructs a random graph with 10 vertices and with each edge as likely to appear as not.

```
In[242]:= RandomGraph[BernoulliGraphDistribution[10, .5]]
```

Out[242]=



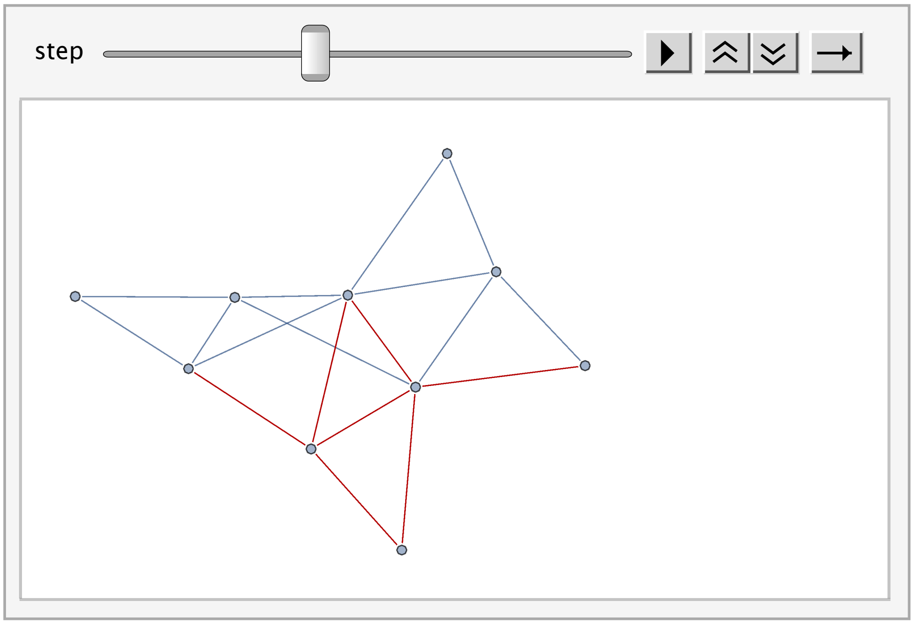
`RandomGraph` also accepts an optional second argument, a positive integer, causing it to produce a list of that many random graphs.

Recall the description of the `EulerianGraphQ` function. When this function is given a graph, it returns true or false depending on the existence of an Euler circuit.

To satisfy the requirements of this problem, we use `RandomGraph` to generate a random graph G. Then we test it for an Euler circuit using `EulerianGraphQ`. As long as the randomly generated graph does not have an Euler circuit, we continue generating new random graphs. We display the path using `FindEulerianCycle` and the **animatePath** function we created in Section 10.5.

```
In[236]:= generateEulerian[n_Integer] /; n > 0 := Module[{G, path},
             While[! EulerianGraphQ[G],
              G = RandomGraph[BernoulliGraphDistribution[n, .5]]];
             animatePath[G, First[FindEulerianCycle[G]]]
            ]
```

```
In[237]:= generateEulerian[10]
```



Computations and Explorations 13

> Estimate the probability that a randomly generated simple graph with *n* vertices is connected for each possible integer *n* not exceeding ten by generating a set of random simple graphs and determining whether each is connected.

Solution: To solve this problem we will create a function that generates a number of random graphs of the specified size and counts the number that are connected. We use the `RandomGraph` function to create the random graphs and the `ConnectedGraphQ` function to test them for connectivity.

```
In[238]:= connectedProbability[n_Integer, max_Integer] /;
             n > 0 && max > 0 := Module[{G, i, count = 0},
             For[i = 1, i ≤ max, i++,
              G = RandomGraph[BernoulliGraphDistribution[n, .5]];
              If[ConnectedGraphQ[G], count++]
             ];
             count / max
            ]
```

```
In[239]:= Table[connectedProbability[i, 100], {i, 10}]
```

$$\text{Out[239]=}\ \left\{1, \frac{12}{25}, \frac{14}{25}, \frac{31}{50}, \frac{69}{100}, \frac{77}{100}, \frac{22}{25}, \frac{97}{100}, \frac{19}{20}, 1\right\}$$

Exercises

1. Write a *Mathematica* function to find *all* maximal matchings for a bipartite graph.
2. Write *Mathematica* functions for calculating the adjacency and incidence matrices for a pseudograph.
3. Write a *Mathematica* function for creating a pseudograph from an incidence matrix.
4. Write a *Mathematica* function to find all of the minimal edge cuts of a given graph.
5. Write a *Mathematica* function to count the number of Hamilton circuits in a simple graph.
6. Write a *Mathematica* function to determine whether a mixed graph (with directed edges, multiple edges, and loops) has an Euler circuit and, if so, to find such a circuit.
7. Use *Mathematica* to construct all regular graphs of degree n, given a positive integer n. (Regular is defined in the Exercises for Section 10.2.)
8. For vertices u and v in a simple, undirected and connected graph G, the local vertex connectivity $\kappa(u, v)$ is defined to be the minimum number of vertices that must be removed so that there is no path between vertex u and vertex v. Write a *Mathematica* function that calculates the local vertex connectivity of a graph and a pair of its vertices.
9. For vertices u and v in a simple, undirected and connected graph G, the local edge connectivity $\lambda(u, v)$ is defined to be the minimum number of edges that must be removed so that there is no path between vertex u and vertex v. Write a *Mathematica* function that calculates the local edge connectivity of a graph and a pair of its vertices.
10. Write a *Mathematica* function that computes the thickness of a nonplanar simple graph (see the Exercises in Section 10.7 for a definition of thickness).
11. Write a *Mathematica* function for finding an orientation of a simple graph. (An orientation of a graph is defined in the Supplementary Exercises of Chapter 10.)
12. Write a *Mathematica* function for finding the bandwidth of a simple graph. (The bandwidth of a graph is defined in the Supplementary Exercises of Chapter 10.)
13. Write a *Mathematica* function for finding the radius and diameter of a simple graph. (The radius and diameter of a graph are defined in the Supplementary Exercises of Chapter 10.)
14. Use *Mathematica* to find the minimum number of queens controlling an $n \times n$ chessboard for as many values of n as you can. Make use of the concept of a dominating set, described in the Supplementary Exercises of Chapter 10.
15. Write a *Mathematica* function for finding all self-complementary graphs on n vertices. (A self-complementary graph is a graph which is isomorphic to its own complement.) Use your function to display the self-complementary graphs for as large a n as possible.

16. Write a *Mathematica* function that finds a total coloring for a graph. A total coloring of a graph is an assignment of a color to each vertex and each edge such that: (a) no pair of adjacent vertices have the same color; (b) no two edges with a common endpoint have the same color; and (c) no edge has the same color as either of its endpoints.

17. A sequence of positive integers is called *graphic* if there is a simple graph that has this sequence as its degree sequence. In this context, the degree sequence of a graph is the nondecreasing sequence made up of the degrees of the vertices of the graph. Develop a *Mathematica* function for determining whether a sequence of positive integers is graphic and, if it is, to construct a graph with this degree sequence.