10 Graphs

Introduction

In this chapter we consider ways in which Maple can help you explore and understand graph theory. In particular, we describe how to do computations on graphs using Maple and how Maple 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. Maple 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 Maple does not support pseudographs (or their directed counterparts). So parts of several sections in this chapter are devoted to extending Maple's existing functionality to pseudographs. This will serve to give you tools that you can use to explore these kinds of graphs. More than that, seeing how to create the procedures for pseudographs will also help you to better understand how the procedures work in the slightly "simpler" case of 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.

Maple has a large collection of commands related to graph theory contained in the `GraphTheory` package. In order to access the short forms of these commands, we use the `with` command.

```
> with(GraphTheory):
```

The `GraphTheory` package includes commands that allow us to create new graphs and then add or delete edges and vertices or even contract edges. Subsets of the vertices can be used to induce subgraphs. Some of the commands are used to create special kinds of graphs such as complete graphs, hyper-cubes, the Petersen graph, and random graphs. Other commands compute some of the important characteristics of a given graph, such as its maximum degree, its diameter, or its planarity.

To create a new graph, we use the `Graph` command. There are a variety of forms of the `Graph` command, but the most natural uses two arguments: a list of vertices and a set of edges. The edges are given as either sets or lists (e.g., `{1,2}` or `[1,2]`) depending on whether the graph is undirected or directed. We demonstrate the creation of a graph by constructing the graph in Exercise 3 in Section 10.1.

```
> Exercise3 := Graph(["a","b","c","d"],
                     {{"a","b"},{"a","c"},{"b","c"},{"b","d"}});
  Exercise3 := Graph 41: an undirected unweighted graph with 4 vertices and 4 edge(s)      (10.1)
```

Note that Maple always expects the vertices to be given in a list and the edges in a set. This is how Maple differentiates between them in alternate forms of the command.

The `Vertices` and `Edges` commands can be used to recover the vertices and edges of a graph.

```
> Vertices(Exercise3);
                    ["a", "b", "c", "d"]                                    (10.2)
> Edges(Exercise3);
          {{"a", "b"}, {"a", "c"}, {"b", "c"}, {"b", "d"}}                  (10.3)
```

Note that explicitly specifying the vertices when defining a graph is often unnecessary, as Maple can determine the vertices from the definition of the edges. The easiest way to define a graph is to call `Graph` with only one argument, the set of edges.

```
> Ex3again := Graph({{"a","b"},{"a","c"},{"b","c"},{"b","d"}});
  Ex3again := Graph 42: an undirected unweighted graph with 4 vertices and 4 edge(s)   (10.4)
> Vertices(Ex3again);  Edges(Ex3again);
                    ["a", "b", "c", "d"]
          {{"a", "b"}, {"a", "c"}, {"b", "c"}, {"b", "d"}}                  (10.5)
```

We can modify existing graphs by adding and deleting edges and vertices using the commands `AddEdge`, `AddArc`, `AddVertex`, `DeleteEdge`, `DeleteArc`, and `DeleteVertex`. (Note: Maple refers to a directed edge as an arc, so the edge commands are used in the case of an undirected graph, and the arc commands are used for directed graphs.)

First we add two vertices to `Ex3again`.

```
> Ex3plus := AddVertex(Ex3again,["y","z"]);
  Ex3plus := Graph 43: an undirected unweighted graph with 6 vertices and 4 edge(s)   (10.6)
> Vertices(Ex3again); Vertices(Ex3plus);
                    ["a", "b", "c", "d"]
                 ["a", "b", "c", "d", "y", "z"]                             (10.7)
```

Now we add edges to connect the new vertices with the rest of the graph.

```
> AddEdge(Ex3plus,{{"a","z"},{"a","y"},{"y","z"}});
        Graph 43: an undirected unweighted graph with 6 vertices and 7 edge(s)      (10.8)
```

And we delete one of the old edges before once again listing the vertices and edges.

```
> DeleteEdge(Ex3plus,{{"b","c"}});
        Graph 43: an undirected unweighted graph with 6 vertices and 6 edge(s)      (10.9)
> Vertices(Ex3plus);  Edges(Ex3plus);
                 ["a", "b", "c", "d", "y", "z"]
   {{"a", "b"}, {"a", "c"}, {"a", "y"}, {"a", "z"}, {"b", "d"}, {"y", "z"}}        (10.10)
```

It is important to be aware of a slight inconsistency in the operation of these commands. The `AddVertex` and `DeleteVertex` commands do not modify the original graph, while the edge and arc commands, by default, do modify the original. The edge commands can be made to not modify the original by giving the equation `inplace=false` as an argument. The vertex commands cannot be made to replace the original except through the usual method of reassignment.

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

```
> Ex3plus := DeleteVertex(Ex3plus,["y"]);
  Ex3plus := Graph 44: an undirected unweighted graph with 5 vertices and 4 edge(s)   (10.11)
> Vertices(Ex3plus);  Edges(Ex3plus);
                    ["a", "b", "c", "d", "z"]
                                                                            (10.12)
```

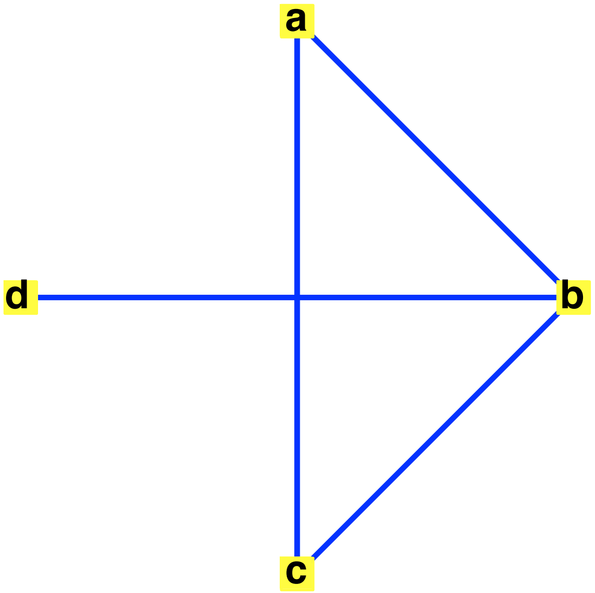
```
{{"a", "b"}, {"a", "c"}, {"a", "z"}, {"b", "d"}}                    (10.12)
```

Visualizing Graphs in Maple

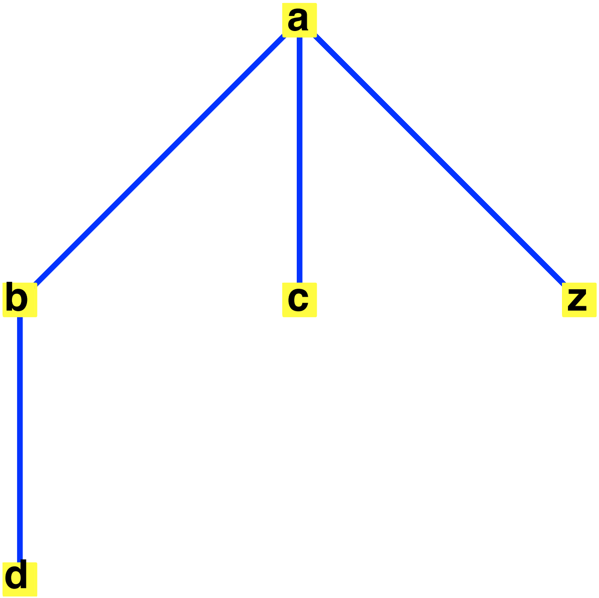
The usefulness of graphs is realized partly through our ability to draw diagrams representing them. Visual representations of graphs sometimes lead to a clearer understanding of the underlying relationships represented by the graphs. The beauty of some of the resulting diagrams is also one of the things that helps to make this such a popular subject.

In Maple, we present a graph visually using the `DrawGraph` command. In its simplest form, this command takes only one argument: the graph to be displayed.

```
> DrawGraph(Exercise3);
```

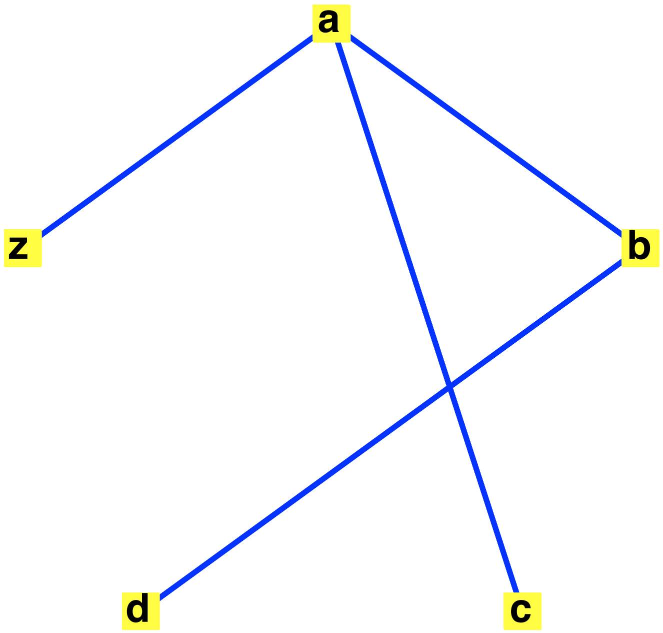


```
> DrawGraph(Ex3plus);
```



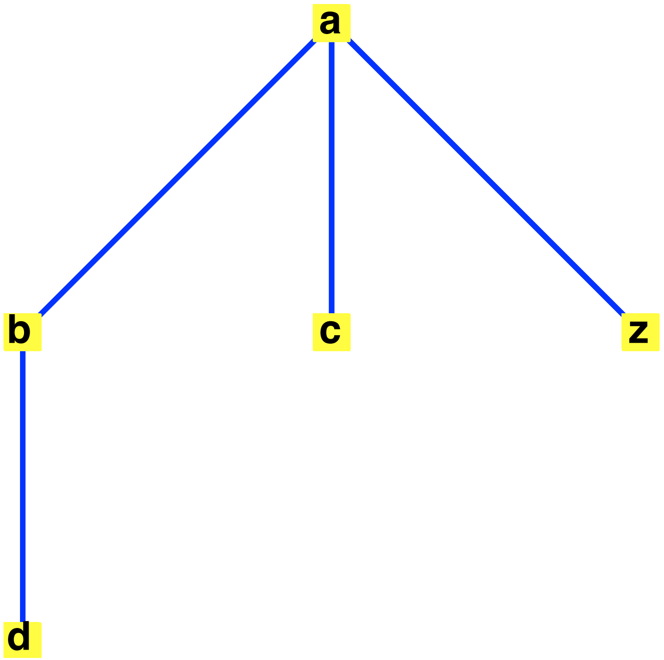
Without any other arguments, Maple does its best to arrange the vertices in reasonable positions. The `style` option allows you to explicitly select one of the types of arrangements. There are five possible styles: `circle`, `tree`, `bipartite`, `spring`, and `planar`. The `circle` style places the vertices on a circle, equally spaced.

```
> DrawGraph(Ex3plus,style=circle);
```



The tree style is available only when the graph is, in fact, a tree — connected and with no circuits (refer to Chapter 11 of the text for more information on trees). `Ex3plus` is a tree. The `DrawGraph` command with no style argument determined that it is a tree and drew it in the tree style automatically. Below, we specify that style explicitly.

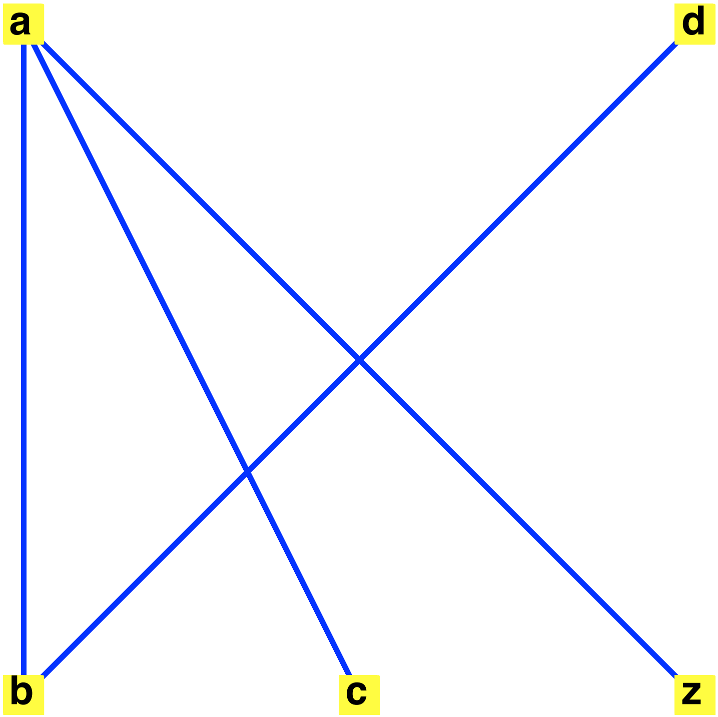
```
> DrawGraph(Ex3plus,style=tree);
```



The `bipartite` style can be used when the graph is bipartite, *i.e.*, the vertices can be separated into two sets such that every edge has one end in one set and the other end in the other (see Section

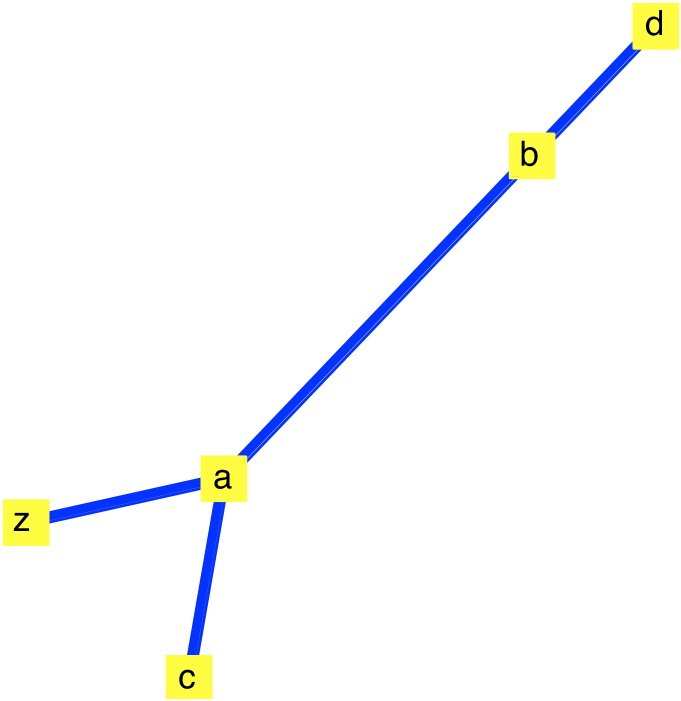
10.2). Maple places the vertices of a bipartite set in two rows indicating the two sets.

```
> DrawGraph(Ex3plus,style=bipartite);
```



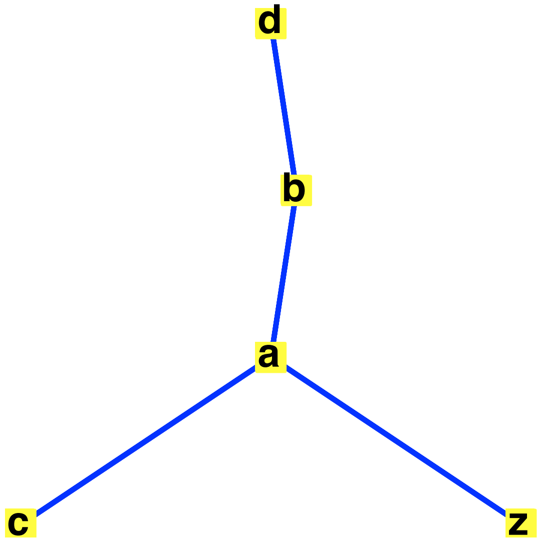
The `spring` style simulates a model where the vertices are taken to be particles repelling each other and the edges are springs pulling vertices together.

```
> DrawGraph(Ex3plus,style=spring);
```



Finally, the `planar` style attempts to draw the graph as a planar graph, *i.e.*, with no edges crossing each other (see Section 10.7 for more on planar graphs). If the graph is nonplanar, then the command will result in an error.

```
> DrawGraph(Ex3plus,style=planar);
```



Pseudographs: Loops and Multiple Edges

As mentioned above, Maple's `GraphTheory` package does not support graphs that have loops or multiple edges. For example, if we try to add a loop to our `Ex3plus` graph, we get an error.

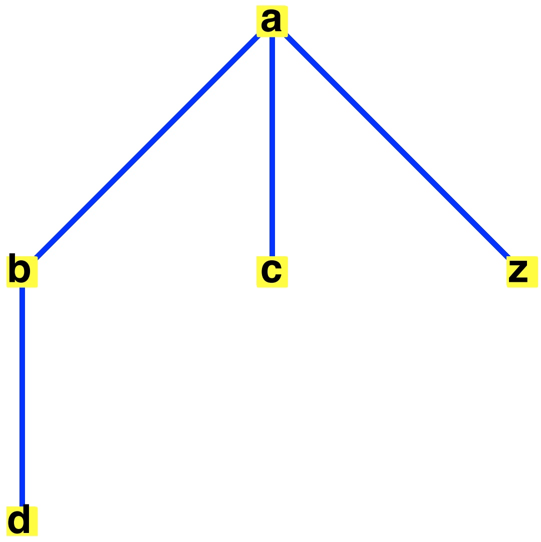
```
> AddEdge(Ex3plus,{{"c","c"}});
Error, (in GraphTheory:-AddEdge) invalid edge {"c"}
```

We also get an error if we try to create a directed graph with a loop.

```
> Graph({["a","b"],["a","a"]});
Error, (in GraphTheory:-Graph) invalid edge/arc: ["a", "a"]
```

On the other hand, multiple edges are merely ignored.

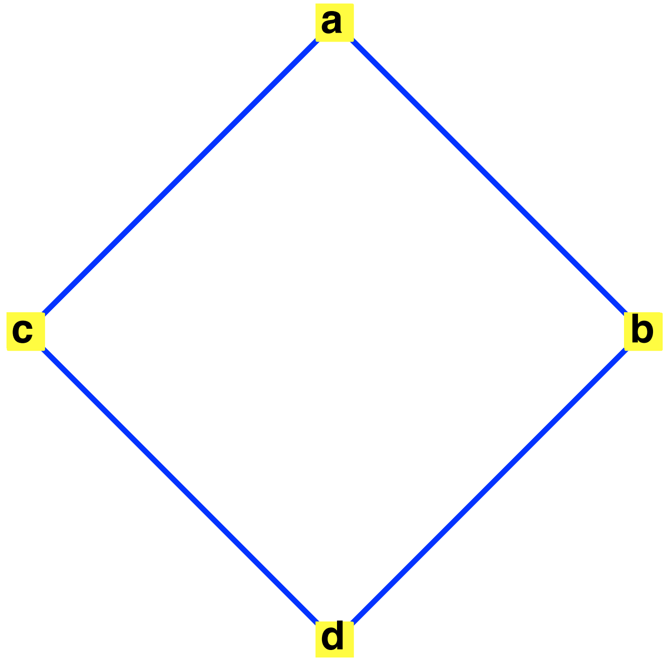
```
> Ex3plus := AddEdge(Ex3plus,{{"a","c"},{"c","a"}});
   Ex3plus := Graph 44: an undirected unweighted graph with 5 vertices and 4 edge(s)      (10.13)
> Edges(Ex3plus);
                     {{"a", "b"}, {"a", "c"}, {"a", "z"}, {"b", "d"}}                     (10.14)
> DrawGraph(Ex3plus);
```



There are ways to, at least partially, get around these two limitations. By way of illustration, we will attempt to create Exercise 5 from Section 10.1. We begin with the simple version of the graph, that is, the graph with the loops and multiple edges omitted.

```
> Exercise5:= Graph({{"a","b"},{"a","c"},{"b","d"},{"c","d"}});
  Exercise5 := Graph 45: an undirected unweighted graph with 4 vertices and 4 edge(s)    (10.15)
```

```
> DrawGraph(Exercise5,style=planar);
```



Loops

With regards to loops, we can mark vertices as having a loop by setting an attribute. An attribute can be used to store arbitrary information about a vertex (or an edge, or for the graph as a whole) in the form `tag=value`. The tag and value can be nearly anything at all. In this case, we'll use the tag "loop" and the value will be true or false.

The `SetVertexAttribute` command takes three arguments: the name of the graph, the name of the vertex, and the attribute in the `tag=value` format.

```
> SetVertexAttribute(Exercise5,"a","loop"=true);
> SetVertexAttribute(Exercise5,"b","loop"=true);
> SetVertexAttribute(Exercise5,"d","loop"=true);
```

We check the value of an attribute with the `GetVertexAttribute` command, which accepts a graph, a vertex, and the tag whose value is desired. Note that if the attribute has not been set, this returns `FAIL`. You can also list all the tags set for a given vertex using the `ListVertexAttributes` command. Note that this does not display the values, only the tags.

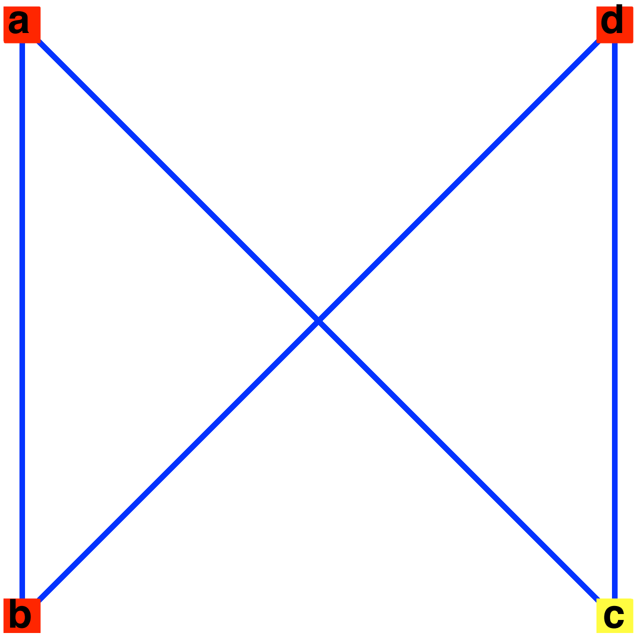
```
> GetVertexAttribute(Exercise5,"a","loop");
                                   true                                   (10.16)
> GetVertexAttribute(Exercise5,"c","loop");
                                   FAIL                                   (10.17)
> ListVertexAttributes(Exercise5,"b");
                         ["draw-pos-planar", "loop"]                      (10.18)
```

We will use attributes to write a program that marks the vertices that have loops by changing their color to red. The color change will be done with the `HighlightVertex` command. The `HighlightVertex` command requires two arguments: the name of the graph and a vertex or list or set of vertices to be highlighted. It optionally accepts a color to use as the highlight.

Here is the procedure to highlight loops.

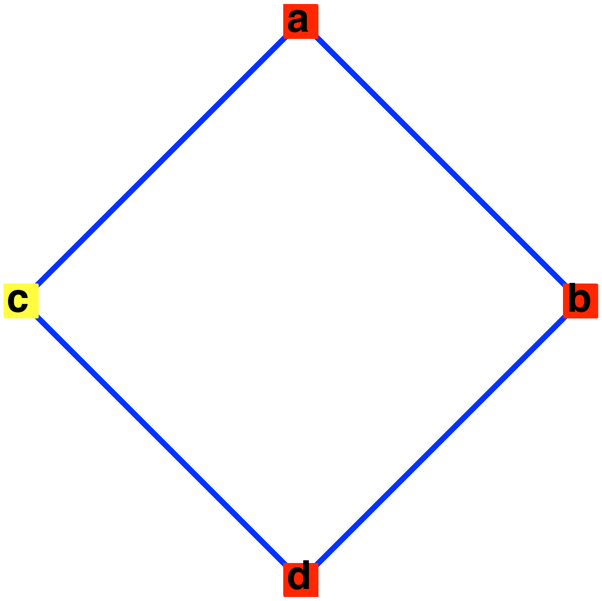
```
> DrawLoops := proc(G::Graph)
    local v;
    uses GraphTheory;
    for v in Vertices(G) do
      if GetVertexAttribute(G,v,"loop") then
        HighlightVertex(G,v,red);
      end if;
    end do;
    DrawGraph(G);
  end proc:
```

```
> DrawLoops(Exercise5);
```



Note that the color changes remain in effect until they are changed again. So, if we draw the graph in the planar style, the vertices with loops remain red.

```
> DrawGraph(Exercise5,style=planar);
```



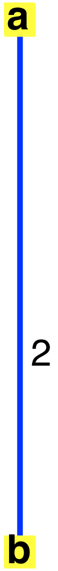
You might not be surprised to discover that, in fact, **HighlightVertex** sets a vertex attribute.

```
> ListVertexAttributes(Exercise5,"a");
        ["draw-pos-planar", "loop", "draw-vertex-color", "draw-pos-default"]          (10.19)
> GetVertexAttribute(Exercise5,"a","draw-vertex-color");
                        COLOUR(RGB, 1.00000000, 0., 0.)                                (10.20)
```

Multiple edges

We now turn our attention to the representation of multiple edges, which we will do with edge weights.

```
> Graph({[{"a","b"},2]});
        Graph 46: an undirected weighted graph with 2 vertices and 1 edge(s)          (10.21)
> DrawGraph((10.21));
```



Note that the edge above is specified as the list `[{"a","b"},2]`. This list consists of two elements: first is the set consisting of the endpoints of the edge, and second is the weight of the edge. `DrawGraph` displays the edge weight next to the edge.

For an existing graph, we can assign weights to edges with the `SetEdgeWeight` command, which takes as arguments the name of the graph, the edge to be weighted, and the weight. It returns the previous weight of the edge. The `SetEdgeWeight` command can only be used with a graph that Maple considers to be weighted. To add weights to an unweighted graph, we first must use the `MakeWeighted` command. (Note, `MakeWeighted` does not change the original graph, it creates a weighted copy of the graph.)

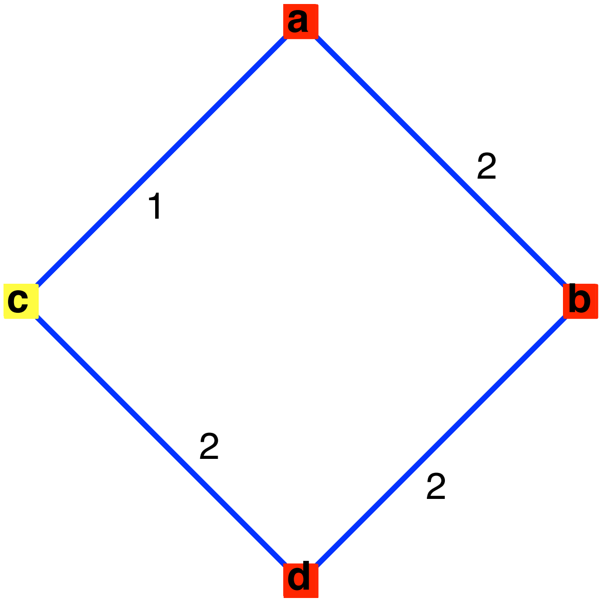
```
> Exercise5 := MakeWeighted(Exercise5);
   Exercise5 := Graph 47: an undirected weighted graph with 4 vertices and 4 edge(s)      (10.22)
```

```
> SetEdgeWeight(Exercise5,{"a","b"},2);
                                   1                                                  (10.23)
```

```
> SetEdgeWeight(Exercise5,{"b","d"},2);
                                   1                                                  (10.24)
```

```
> SetEdgeWeight(Exercise5,{"c","d"},2);
                                   1                                                  (10.25)
```

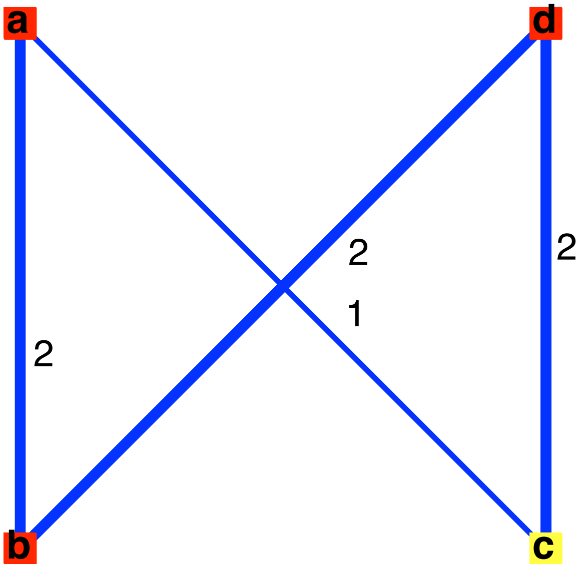
```
> DrawGraph(Exercise5,style=planar);
```



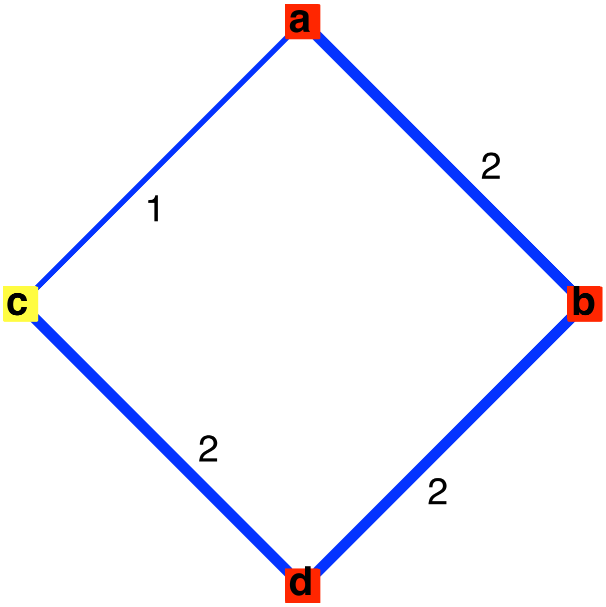
The "draw-edge-thickness" edge attribute increases the thickness of the line representing an edge. Let's expand on our `DrawLoops` procedure above to not only set the color of vertices with loops but to also give a visual representation of multiple edges by thickening them. We set the thickness of edges to $3n - 2$ where n is the weight of the edge (*i.e.*, the number of edges) so that a single edge has thickness 1, and each additional edge increases the thickness by 3.

```
> DrawPseudograph := proc(G::Graph)
    local v, e, w;
    uses GraphTheory;
    for v in Vertices(G) do
      if GetVertexAttribute(G,v,"loop") then
        HighlightVertex(G,v,red);
      end if;
    end do;
    if IsWeighted(G) then
```

```
      for e in Edges(G) do
        if GetEdgeWeight(G,e) > 1 then
          w := 3*GetEdgeWeight(G,e) - 2;
          SetEdgeAttribute(G,e,
               "draw-edge-thickness"=w);
        end if;
      end do;
    end if;
    DrawGraph(G);
  end proc:
> DrawPseudograph(Exercise5);
```



```
> DrawGraph(Exercise5,style=planar);
```



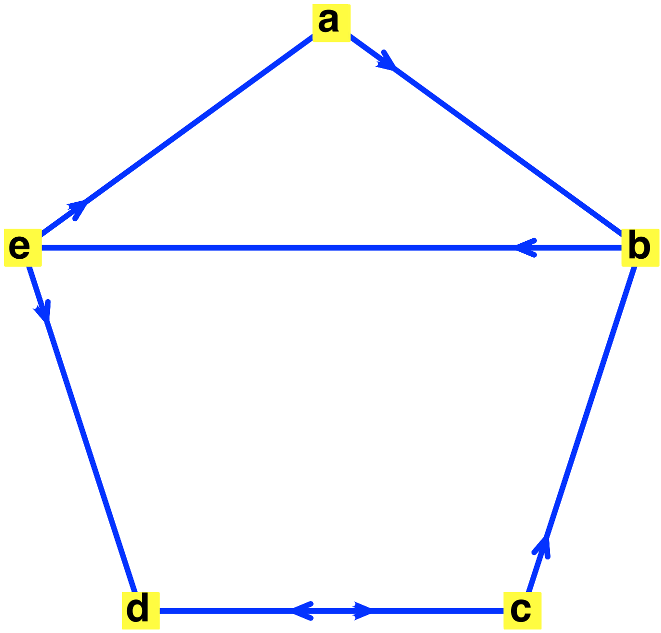
Directed Graphs

Finally, let's consider an example of a directed graph. Specifically, we will reproduce, as far as possible, Exercise 7 from Section 10.1. We will create this graph with the **Digraph** command, which works like `Graph` but emphasizes that the graph is directed.

We also use the `Trail` command in this example. This command is used to specify a sequence of edges. For instance, `Trail(1,2,3,1)` is a shorter way to specify the edges `[1,2]`, `[2,3]`, and `[3,1]`. The `Digraph` and `Graph` commands allow us to include applications of `Trail` alongside a set containing additional edges.

```
> Exercise7 := Digraph(["a","b","c","d","e"],
                        Trail("e","a","b","e","d","c","b"),
                        {["c","d"]});
   Exercise7 := Graph 48: a directed unweighted graph with 5 vertices and 7 arc(s)      (10.26)
```

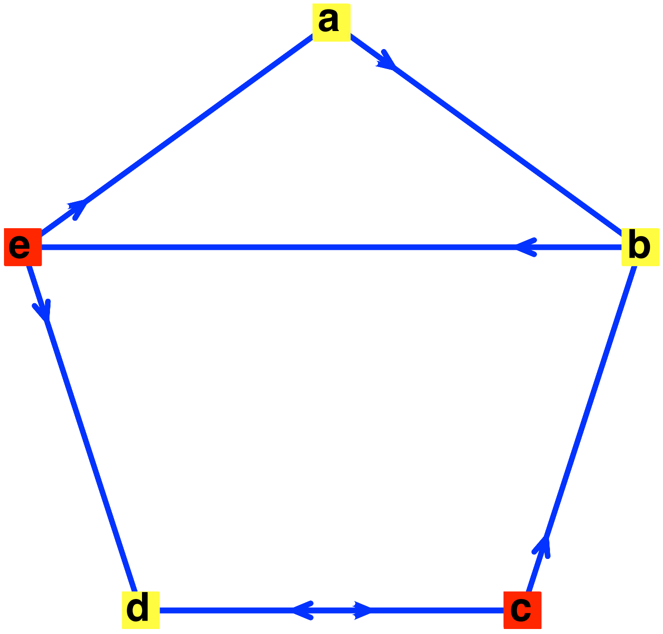
```
> DrawGraph(Exercise7);
```



Notice that the edge between *c* and *d* has two arrows representing the pair of edges between them.

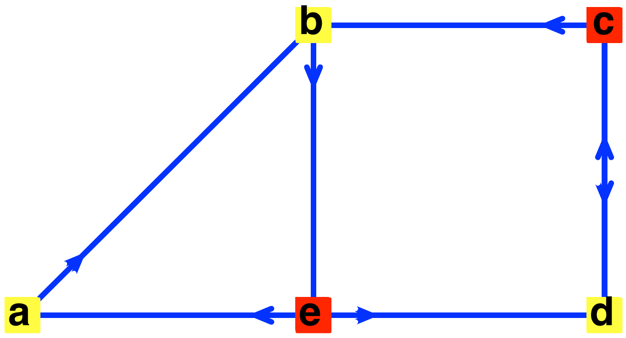
Now we add the loops.

```
> SetVertexAttribute(Exercise7,"c","loop"=true);
> SetVertexAttribute(Exercise7,"e","loop"=true);
> DrawLoops(Exercise7);
```



While this image displays all of the information contained in the graph, the position of the vertices makes it look very different from the drawing in the textbook. We can override Maple's choice of vertex position with the `SetVertexPositions` command. This command takes two arguments: the graph and a list of pairs specifying the x and y coordinates of each vertex. The first pair specifies the location of the first vertex, the second pair the second vertex, *etc*.

```
> SetVertexPositions(Exercise7,[[0,0],[1,1],[2,1],[2,0],[1,0]])
  ;
> DrawGraph(Exercise7);
```



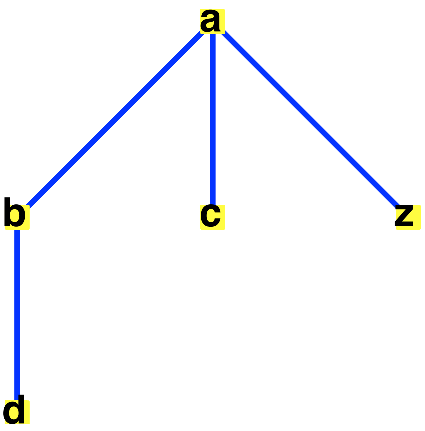
10.2 Graph Terminology and Special Types of Graphs

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

Degree

Maple's `GraphTheory` package has a `Degree` function for determining the degree of a vertex. Given a graph 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 *z* of `Ex3plus` from above.

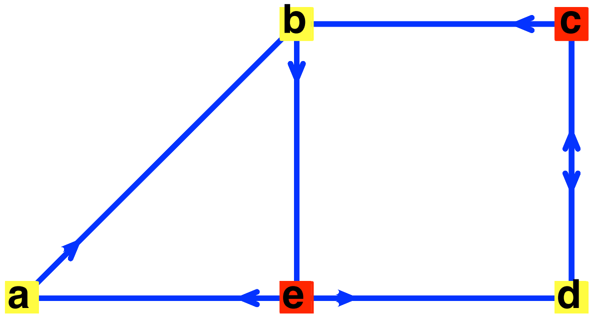
```
> DrawGraph(Ex3plus);
```



```
> Degree(Ex3plus,"a");
                              3                                  (10.27)
> Degree(Ex3plus,"z");
                              1                                  (10.28)
```

For a directed graph, the `Degree` command calculates the number of edges incident to the given vertex without regard for their direction. But Maple also provides `InDegree` and `OutDegree` commands for calculating the directed values. As an example, consider vertex *d* in the `Exercise7` graph from the previous section.

```
> DrawGraph(Exercise7);
```



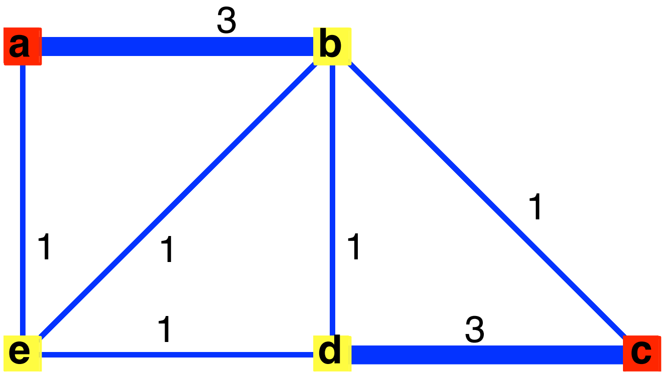
```
> InDegree(Exercise7,"d");
                                   2                                  (10.29)
> OutDegree(Exercise7,"d");
                                   1                                  (10.30)
```

Degree in pseudographs

Note that Maple's built-in commands for degree cannot take into account loops or multiple edges. We will write a procedure to rectify this, at least for pseudographs (undirected graphs which may have loops and multiple edges). The procedures for directed graphs are left to the reader.

First, we reproduce Exercise 2 from Section 10.2 to use as an example. Note that for the multiple edges, we use the weighted edge format `[{v1,v2},w]` which indicates an undirected edge between `v1` and `v2` with weight `w`. For the single edges, we use the usual `{v1,v2}` format. The presence of weighted edges tells Maple that the graph is weighted and causes it to assign weight 1 to edges that are not given a specific weight.

```
> Exercise2 := Graph({[{"a","b"},3],{"a","e"},{"b","c"},{"b",
  "d"},{"b","e"},[{"c","d"},3],{"d","e"}});
  Exercise2 := Graph 49: an undirected weighted graph with 5 vertices and 7 edge(s)   (10.31)
> SetVertexPositions(Exercise2,[[0,1],[1,1],[2,0],[1,0],[0,0]])
  ;
> SetVertexAttribute(Exercise2, "a", "loop"=true);
> SetVertexAttribute(Exercise2, "c", "loop"=true);
> DrawPseudograph(Exercise2);
```



To calculate the degree of a vertex, we first check to see if the graph is weighted or not using the IsWeighted command. If it is not weighted (*i.e.*, there are no multiple edges), we just use the built-in Degree function. If it is weighted, then we use the command IncidentEdges to determine the edges incident to the given vertex. The add command adds up the weights for us. Then we just have to check to see if there is a loop and, if so, add 2 to the degree.

```
> PseudoDegree := proc(G::Graph, v)
```

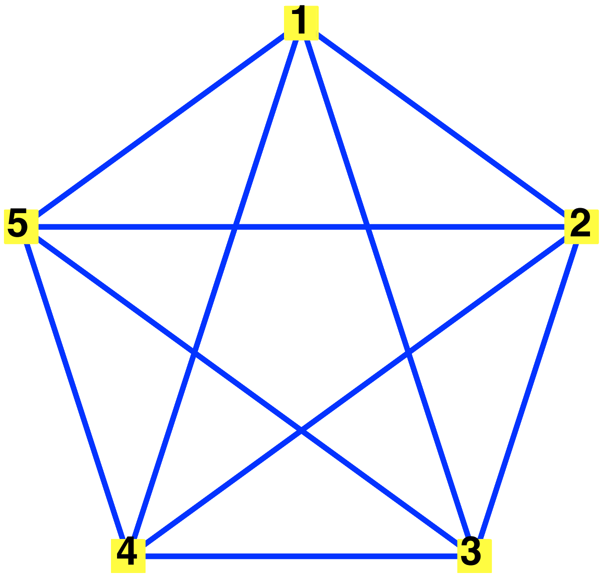
```
    local E, e, d;
    uses GraphTheory;
    if IsWeighted(G) then
      d := add(GetEdgeWeight(G,e),e in IncidentEdges(G,v));
    else
      d := Degree(G,v);
    end if;
    if GetVertexAttribute(G,v,"loop") then
      d := d + 2;
    end if;
    return d;
  end proc:
> PseudoDegree(Exercise2,"a");
                                   6                                   (10.32)
> PseudoDegree(Exercise2,"d");
                                   5                                   (10.33)
```

Some Special Simple Graphs

The textbook discusses several families of graphs, including complete graphs, cycles, wheels, and *n*-cubes. Maple provides commands for easily creating these and other special simple graphs.

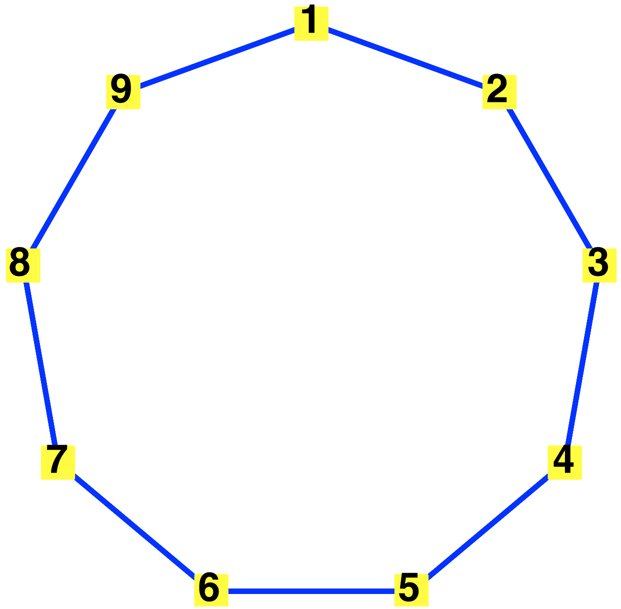
We begin with complete graphs. Recall that a complete graph is a simple, undirected graph on a given number of vertices that has all possible edges between those vertices. The complete graph on n vertices is denoted K_n. The function `CompleteGraph` generates complete graphs. For example, we can generate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

```
> DrawGraph(CompleteGraph(5));
```



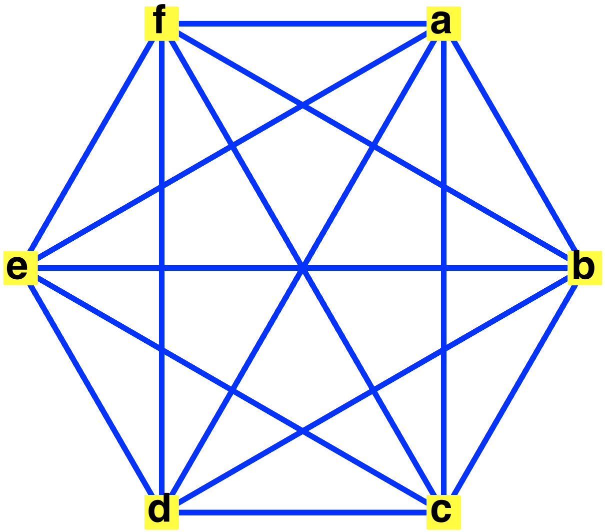
Similarly, we may construct a cycle C_n with the `CycleGraph` command.

```
> DrawGraph(CycleGraph(9));
```



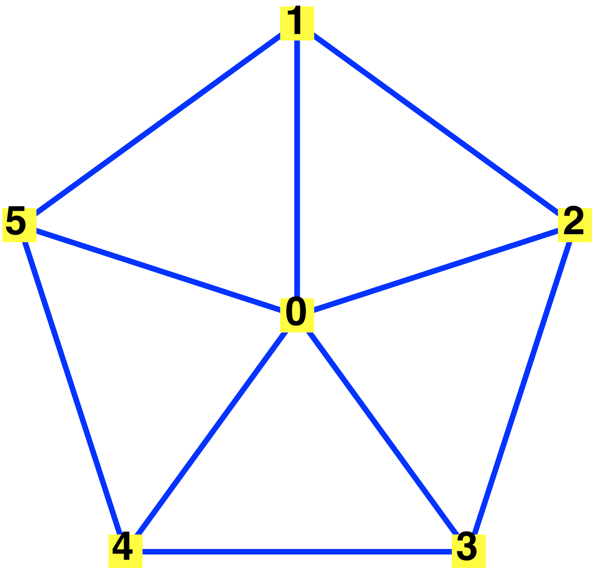
For both the complete graphs and cycle graphs, if you prefer the vertices to be labeled with something other than the integers 1 through *n*, you can call the commands with a list of vertices instead.

```
> DrawGraph(CompleteGraph(["a","b","c","d","e","f"]));
```



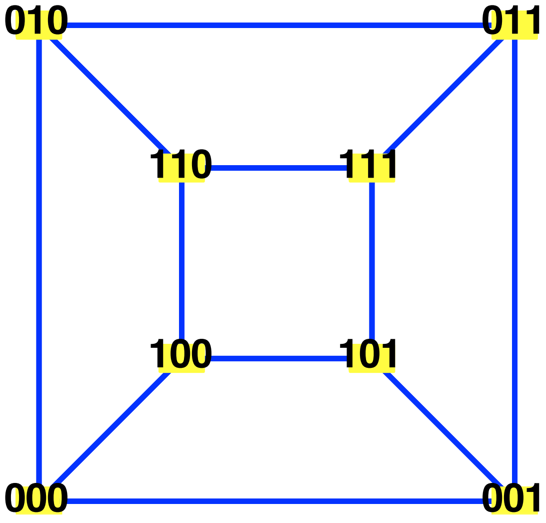
A wheel W_n is obtained from the cycle graph C_n by adding one additional vertex adjacent to all *n* of the original vertices. In Maple, the `WheelGraph` command is part of the `SpecialGraphs` package.

```
> DrawGraph(SpecialGraphs[WheelGraph](5));
```



To construct the n-cube Q_n, we use the **HypercubeGraph** command. 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.

```
> DrawGraph(SpecialGraphs[HypercubeGraph](3));
```



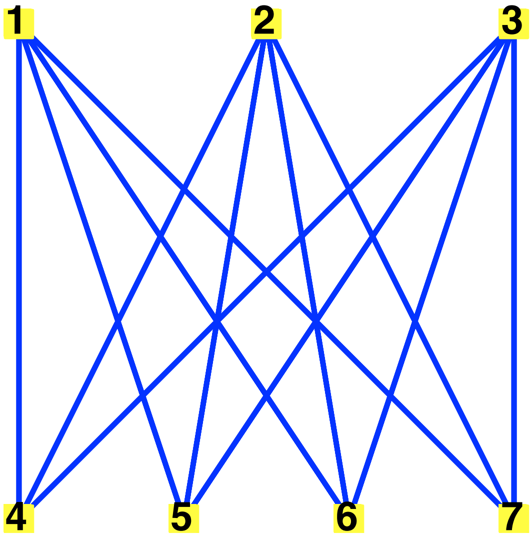
The help page for the **SpecialGraphs** package lists all of the available graphs. The reader is encouraged to spend some time exploring that package.

Bipartite Graphs

Another important class of graphs is the class of 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** command that was discussed earlier can be used to create complete bipartite graphs by entering the two integers m and n.

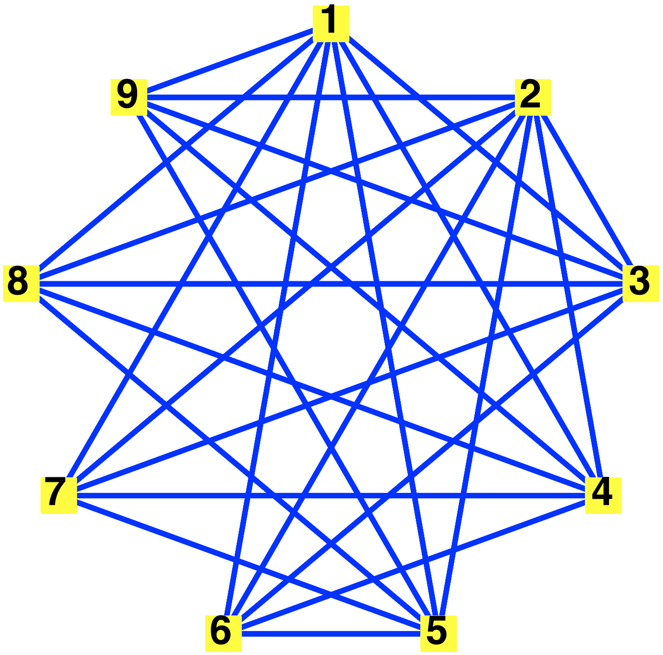
```
> DrawGraph(CompleteGraph(3,4));
```



Notice that Maple draws the complete bipartite graph with the two partitioning sets $\{1,2,3\}$ and $\{4,5,6,7\}$ along the top and bottom of the graph, respectively, to make the partition visually clear.

Maple 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.

```
> DrawGraph(CompleteGraph(2,3,4));
```



Maple has a built in command for determining whether a graph is bipartite: **IsBipartite**. This command takes two arguments: the name of a graph and an unused name in which the bipartition is stored in the case the graph is bipartite.

```
> IsBipartite(SpecialGraphs[HypercubeGraph](3),'examplePart');
                                   true                                    (10.34)
> examplePart;
          [["000", "011", "101", "110"], ["001", "010", "100", "111"]]       (10.35)
```

It is worthwhile, however, recreating a version of **IsBipartite** from scratch in order to better understand the algorithm that determines whether the graph is bipartite and finds a bipartition. Instead of just returning true, our procedure 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 procedure will return false.

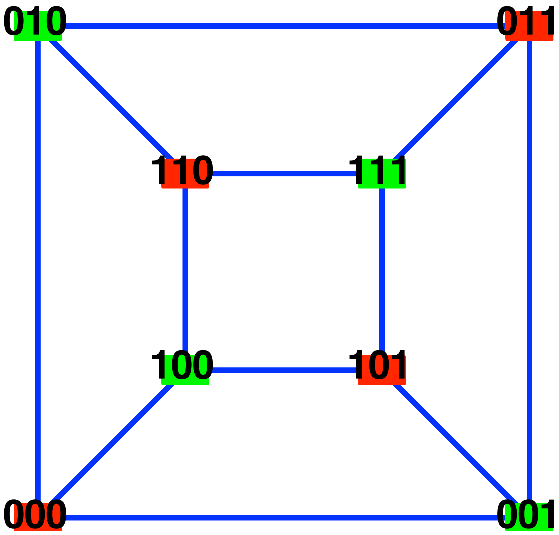
The idea of the procedure is as follows. (Note that this method is based on forming a spanning tree of the graph, a concept discussed in Section 11.4 of the textbook).

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.

Here is the implementation of our procedure **DrawBipartite**.

```
> DrawBipartite := proc(G::Graph)
    local V, E, AB, i, T, w, e;
    uses GraphTheory;
    V := {op(Vertices(G))};
    E := Edges(G);
    w := V[1];
    AB[0] := {w};
    AB[1] := {};
    i := 0;
    while V <> {} do
      T := V intersect AB[i];
      i := i + 1 mod 2;
      for w in T do
        AB[i] := AB[i] union
             ({op(Neighbors(G,w))} minus (AB[0] union AB[1]));
      end do;
      V := V minus T;
    end do;
    for e in E do
      if ((e[1] in AB[0]) and (e[2] in AB[0])) or
         ((e[1] in AB[1]) and (e[2] in AB[1])) then
        return false;
      end if;
    end do;
    HighlightVertex(G,AB[0],red);
    HighlightVertex(G,AB[1],green);
    DrawGraph(G);
  end proc:
```

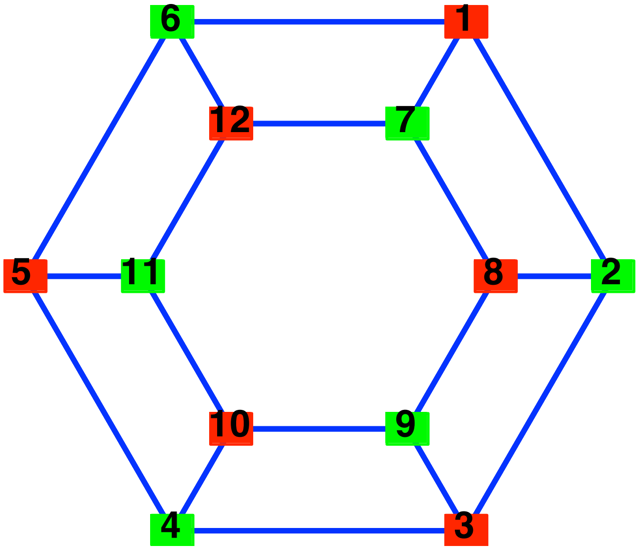
```
> DrawBipartite(SpecialGraphs[HypercubeGraph](3));
```



```
> DrawBipartite(CompleteGraph(6));
```

false **(10.36)**

```
> DrawBipartite(SpecialGraphs[PrismGraph](6));
```



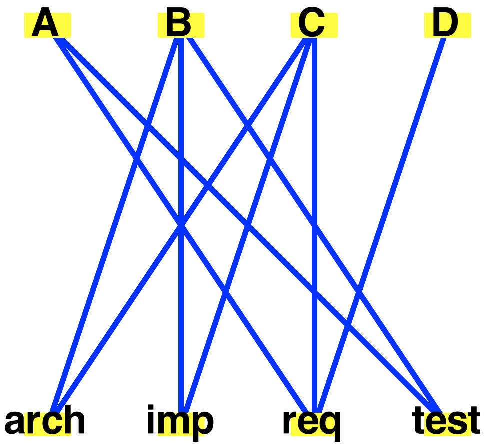
Bipartite Graphs and Matchings

Maple 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.

```
> Figure10a := Graph({{"A","req"},{"A","test"},{"B","arch"},
  {"B","imp"},{"B","test"},{"C","req"},{"C","arch"},{"C","imp"}
  ,{"D","req"}});
```

Figure10a := Graph 50: an undirected unweighted graph with 8 vertices and 9 edge(s) **(10.37)**

```
> DrawGraph(Figure10a);
```



To find a maximal matching, we use the command **BipartiteMatching**. The only allowed argument to this command is the name of the graph. It returns a sequence with two elements: (1) the size of a maximal matching, that is, the largest possible number of edges in a matching; and (2) a set of edges which forms one maximal matching.

```
> Figure10aResult := BipartiteMatching(Figure10a);
  Figure10aResult := 4, {{"A", "test"}, {"B", "arch"}, {"C", "imp"}, {"D", "req"}}     (10.38)
```

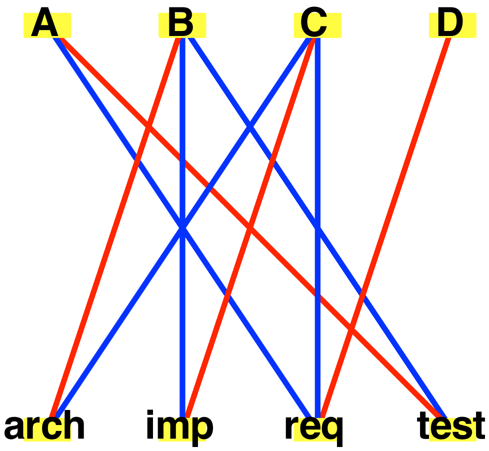
The above output indicates that one maximal matching has Alvarez assigned to testing, Berkowitz to architecture, Chen to implementation, and Davis to requirements. (Note that Maple has produced a different matching than the one given in the text.)

We can visualize this matching by having Maple highlight the edges that form the matching. The edges of the matching are the second element in the output, so we access the set of edges as follows.

```
> Figure10aResult[2];
                {{"A", "test"}, {"B", "arch"}, {"C", "imp"}, {"D", "req"}}                (10.39)
```

We use that as the second argument to the **HighlightEdges** command. Given a graph and a set of edges, **HighlightEdges** changes the color of the specified edges. It can also take an optional third argument specifying the color to use.

```
> HighlightEdges(Figure10a,Figure10aResult[2]);
> DrawGraph(Figure10a);
```



Subgraphs and Induced Subgraphs

Maple provides two primary methods for creating subgraphs. The **Subgraph** command takes two arguments: a graph and a list of edges. It returns the graph whose edge set is the given set of edges and whose vertex set is the vertices that are an endpoint of one of the given edges.

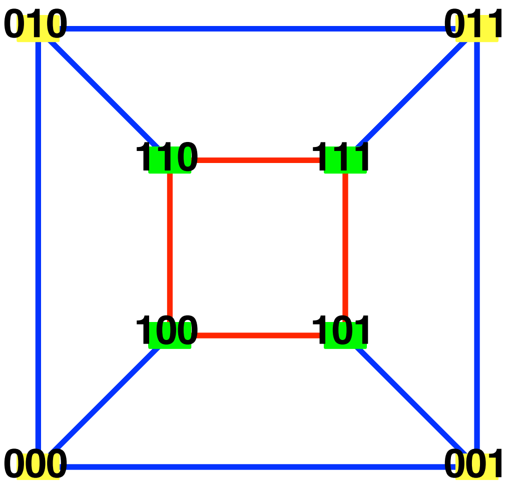
```
> hyper := SpecialGraphs[HypercubeGraph](3);
   hyper := Graph 51: an undirected unweighted graph with 8 vertices and 12 edge(s)     (10.40)
> subhyper := Subgraph(hyper,{{"100","101"},{"101","111"},
  {"111","110"},{"110","100"}});
  subhyper := Graph 52: an undirected unweighted graph with 4 vertices and 4 edge(s)    (10.41)
> Vertices(subhyper);
                              ["100", "101", "110", "111"]                              (10.42)
> Edges(subhyper);
          {{"100", "101"}, {"100", "110"}, {"101", "111"}, {"110", "111"}}              (10.43)
```

The second method for creating subgraphs is the **InducedSubgraph** command. This command expects a graph and a set (or list) of vertices of the graph. It returns the graph induced by the given vertices, that is, the graph consisting of the given vertices and all the edges from the original graph with endpoints in the set of vertices.

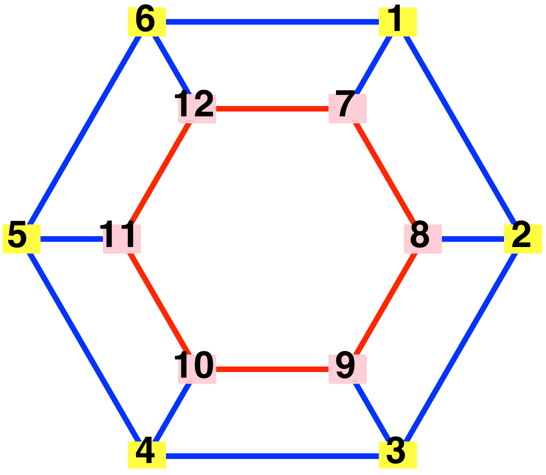
```
> prism := SpecialGraphs[PrismGraph](6);
  prism := Graph 53: an undirected unweighted graph with 12 vertices and 18 edge(s)     (10.44)
> subprism := InducedSubgraph(prism,{7,8,9,10,11,12});
  subprism := Graph 54: an undirected unweighted graph with 6 vertices and 6 edge(s)    (10.45)
> Vertices(subprism);
                                  [7, 8, 9, 10, 11, 12]                                 (10.46)
> Edges(subprism);
                  {{7, 8}, {7, 12}, {8, 9}, {9, 10}, {10, 11}, {11, 12}}                (10.47)
```

Maple also provides the command **HighlightSubgraph** to help visualize the structure of a subgraph as part of the original graph. By default, the vertices of the subgraph are set to green and the edges to red. Those color choices can be changed by passing colors to the command.

```
> HighlightSubgraph(hyper,subhyper);
> DrawGraph(hyper);
```



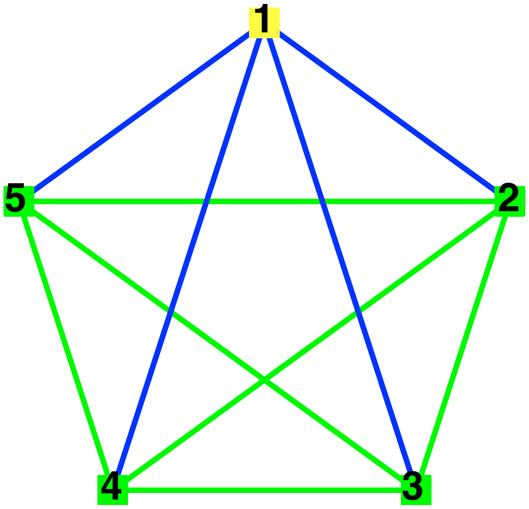
```
> HighlightSubgraph(prism,subprism,red,pink);
> DrawGraph(prism);
```



Deleting Vertices and Edges

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

```
> ExDeleteVStart := CompleteGraph(5);
ExDeleteVStart :=                                                    (10.48)
      Graph 55: an undirected unweighted graph with 5 vertices and 10 edge(s)
> ExDeletedV := DeleteVertex(CompleteGraph(5),1);
ExDeletedV := Graph 56: an undirected unweighted graph with 4 vertices and 6 edge(s)   (10.49)
> HighlightSubgraph(ExDeleteVStart,ExDeletedV,green,green);
> DrawGraph(ExDeleteVStart);
```

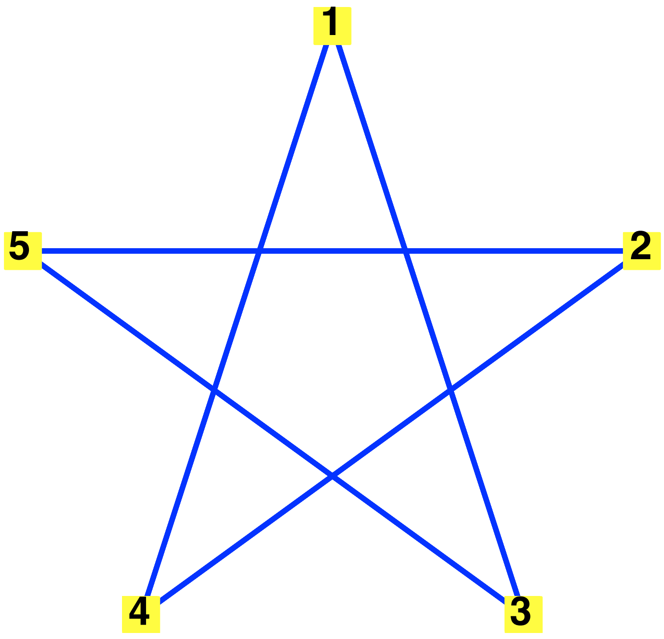


`DeleteEdge` also takes two arguments, a name of an undirected graph and an edge or a set of edges. The set of edges can also be specified as a `Trail`. Recall from above that a `Trail` is a way to specify a set of edges by simply listing all of the vertices, in the order that the sequence of edges pass through them. So, for example, we can specify the outer ring of K_5 as follows.

```
> Trail(1,2,3,4,5,1);
                              Trail(1, 2, 3, 4, 5, 1)                              (10.50)
```

Note that the `Trail` command does not seem to evaluate. This is because the command is inert and only operates as a part of another command. We delete these edges from a complete graph.

```
> ExDeleteE := CompleteGraph(5);
  ExDeleteE := Graph 57: an undirected unweighted graph with 5 vertices and 10 edge(s)  (10.51)
> DeleteEdge(ExDeleteE,Trail(1,2,3,4,5,1));
        Graph 57: an undirected unweighted graph with 5 vertices and 5 edge(s)         (10.52)
> DrawGraph(ExDeleteE);
```

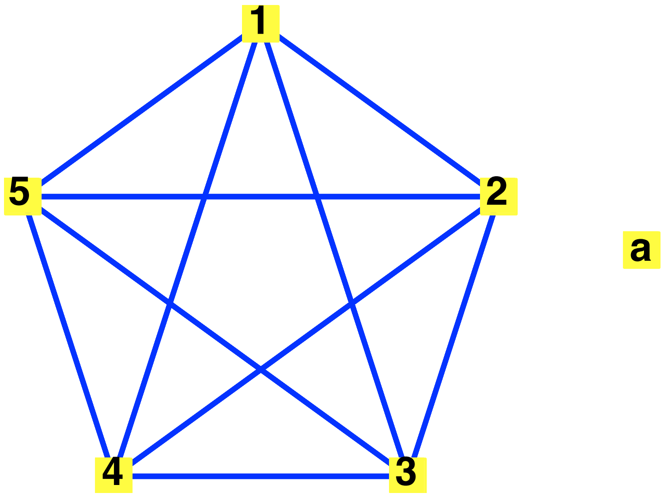


Observe that the `DeleteEdge` command modified its argument, as opposed to the `DeleteVertex` command which did not. To prevent modification of the graph, you can give the `inplace=false` option to `DeleteEdge`.

Adding Vertices and Edges

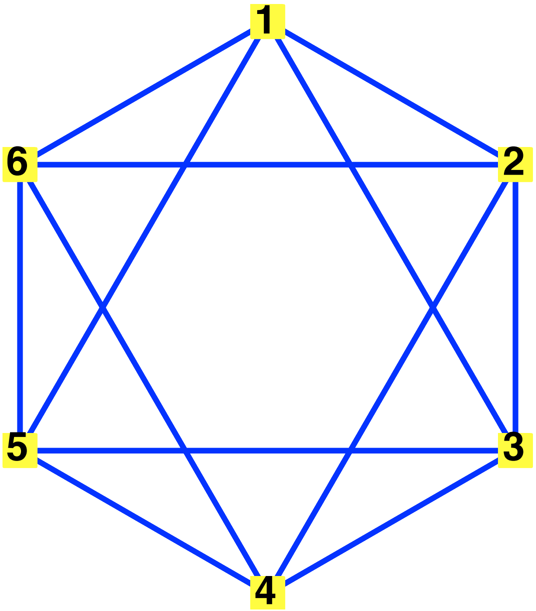
The commands for adding vertices and edges have very similar forms. `AddVertex` accepts a graph and either a vertex or a list of vertices to add to the graph. Again, the original is not modified.

```
> ExAddV := AddVertex(CompleteGraph(5),"a");
  ExAddV := Graph 58: an undirected unweighted graph with 6 vertices and 10 edge(s)    (10.53)
> DrawGraph(ExAddV);
```



`AddEdge` acts on undirected graphs and accepts one edge, a set of edges, or a trail. Also, without the `inplace=false` option, this command will alter the original graph.

```
> ExAddE := CycleGraph(6);
    ExAddE := Graph 59: an undirected unweighted graph with 6 vertices and 6 edge(s)      (10.54)
> AddEdge(ExAddE,{{1,3},{2,4},{3,5},{4,6},{5,1},{6,2}});
        Graph 59: an undirected unweighted graph with 6 vertices and 12 edge(s)           (10.55)
> DrawGraph(ExAddE);
```



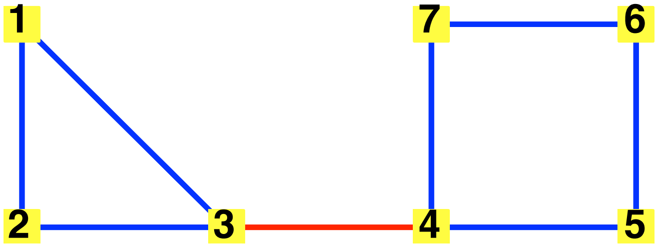
`DeleteEdge` and `AddEdge` apply only for undirected graphs. For directed graphs, use the commands `DeleteArc` and `AddArc`. The syntax and behavior of these commands are exactly the same as their undirected counterparts.

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.

```
> ExContraction := Graph({{1,2},{1,3},{2,3},{3,4},{4,5},{4,7},
  {5,6},{6,7}});
ExContraction :=                                                    (10.56)
      Graph 60: an undirected unweighted graph with 7 vertices and 8 edge(s)
```

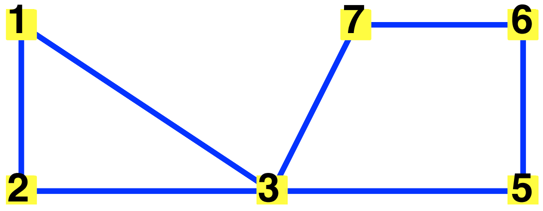
```
> SetVertexPositions(ExContraction,[[0,1],[0,0],[1,0],[2,0],[3,
  0],[3,1],[2,1]]);
> HighlightEdges(ExContraction,{3,4});
> DrawGraph(ExContraction);
```



We will perform an edge contraction on the edge $\{3, 4\}$, which is highlighted in the image. The `Contract` command takes as arguments the name of the graph and an edge and performs the edge contraction. Note that the merged vertices will be represented by one of the vertices in the original pair, in this case vertex 3 will represent the pair. Also note that this command does not modify the original graph.

```
> ExContracted := Contract(ExContraction,{3,4});
ExContracted := Graph 61: an undirected unweighted graph with 6 vertices and 7 edge(s) (10.57)
```

```
> SetVertexPositions(ExContracted,[[0,1],[0,0],[1.5,0],[3,0],
  [3,1],[2,1]]);
> DrawGraph(ExContracted);
```



Unions and Complements of Graphs

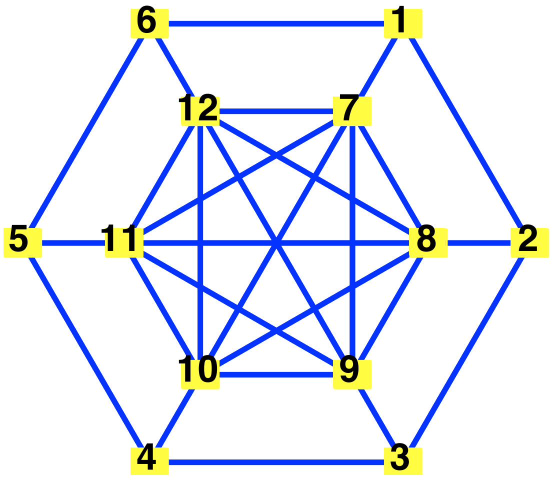
Recall that the union of two graphs is the graph obtained by taking the union of the sets of vertices and the sets of edges from the two graphs.

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 the inner ring.

```
> unionA := SpecialGraphs[PrismGraph](6):
> unionB := CompleteGraph([7,8,9,10,11,12]):
> unionAB := GraphUnion(unionA,unionB):
```

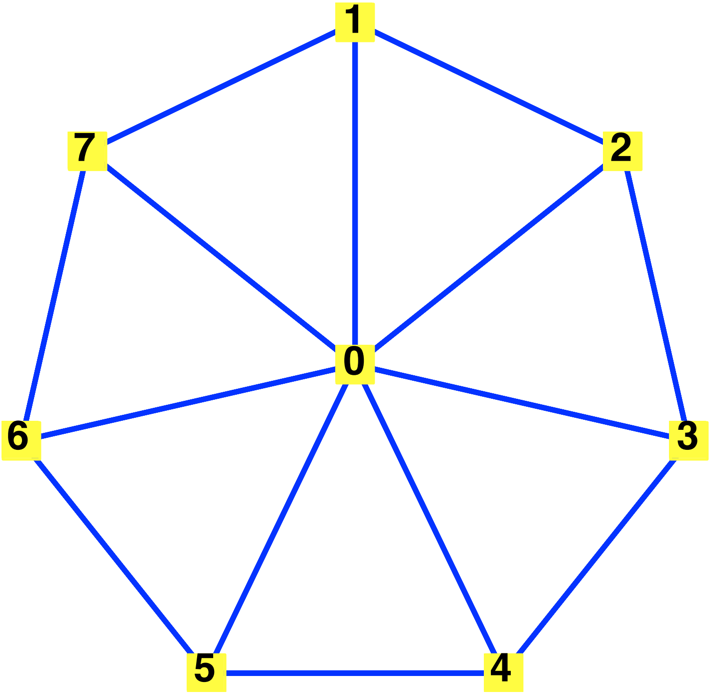
To get this graph to display in the way we described it, as the prism filled in with the complete graph on the inner set of vertices, we need to set the positions of the vertices. Otherwise, Maple will simply draw it in the circular style. We can access the vertex locations used in the display of the prism and impose those locations on this graph.

```
> SetVertexPositions(unionAB,GetVertexPositions(unionA));
> DrawGraph(unionAB);
```

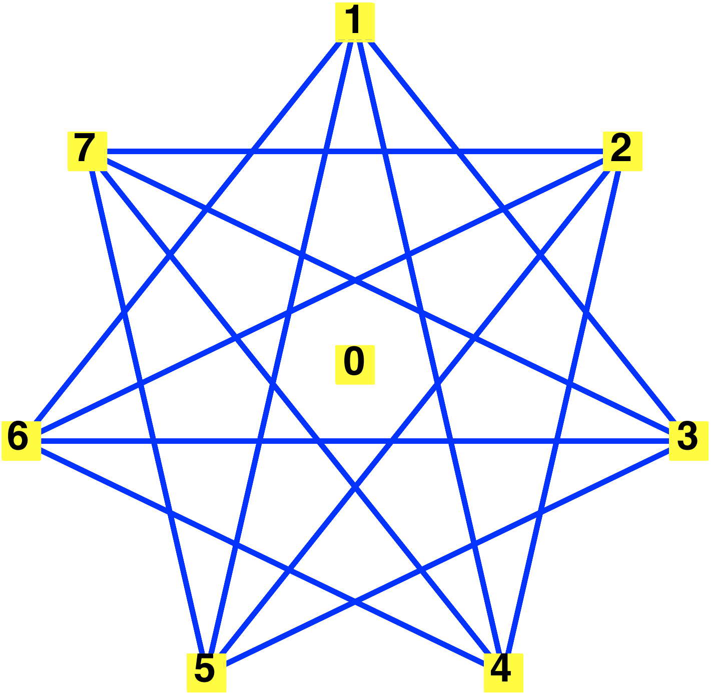


Finally, we consider graph complements, described in Exercise 59 of Section 10.2. The complement, $\overline{G}$, of a graph G is the graph whose vertex set is the same as that of G, but whose edge set is the set of all pairs of G that have no edge between them. In other words, if G has n vertices, then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

```
> DrawGraph(SpecialGraphs[WheelGraph](7));
```



```
> DrawGraph(GraphComplement(SpecialGraphs[WheelGraph](7)));
```



10.3 Representing Graphs and Graph Isomorphism

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

Adjacency Lists

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

In order to define a graph in Maple using an adjacency list, you apply the `Graph` command to a list of sets. For example,

```
> Graph([{2,3},{1,3,4},{1,2},{2,5},{4}]);
        Graph 62: an undirected unweighted graph with 5 vertices and 5 edge(s)        (10.58)
```

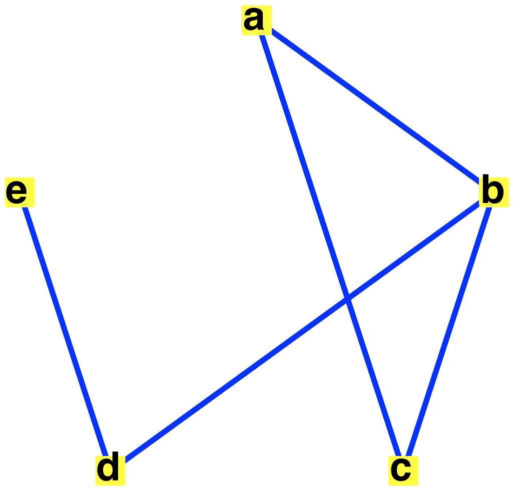
indicates that vertex 1 is incident to vertices 2 and 3; vertex 2 is incident to vertices 1, 3, and 4; vertex 3 is incident to vertices 1 and 2; and so on.

Note that graphs constructed this way are undirected or directed depending on the content of the adjacency lists. For example, in the previous example, removing 1 from the second set would make vertex 1 incident to vertex 2, but not *vice versa*. This would cause Maple to consider the graph directed.

Also note that the vertex labels can be specified by providing a list of the labels, but the adjacency list always needs to consist of integers corresponding to the index of the vertex.

```
> Graph(["a","b","c","d","e"],[{2,3},{1,3,4},{1,2},{2,5},{4}]);
        Graph 63: an undirected unweighted graph with 5 vertices and 5 edge(s)        (10.59)
```

```
> DrawGraph((10.59));
```



In the other direction, we will write a procedure that, given a graph *G*, will print out the adjacency list of each of *G*'s vertices. In order for the procedure to work with both undirected and directed graphs, we will use the Maple command `Departures`. For a directed graph, `Departures(G, v)` returns a list of all of the terminal vertices for edges whose initial vertex is *v*. If *G* is undirected, the same command returns all of *v*'s neighbors. Our procedure will allow for loops by checking the "loop" attribute and listing the vertex as adjacent to itself when the "loop" attribute is true.

```
> AdjacencyLists := proc(G::Graph)
```

```
      local v, AList;
      uses GraphTheory;
      for v in Vertices(G) do
        AList := Departures(G,v);
        if GetVertexAttribute(G,v,"loop") then
          AList := [v, op(AList)];
        end if;
        printf("Vertex %a is adjacent to %a\n",v,AList);
      end do;
    end proc:
> AdjacencyLists(CycleGraph(5));
Vertex 1 is adjacent to [2, 5]
Vertex 2 is adjacent to [1, 3]
Vertex 3 is adjacent to [2, 4]
Vertex 4 is adjacent to [3, 5]
Vertex 5 is adjacent to [1, 4]
> AdjacencyLists(Exercise7);
Vertex "a" is adjacent to ["b"]
Vertex "b" is adjacent to ["e"]
Vertex "c" is adjacent to ["c", "b", "d"]
Vertex "d" is adjacent to ["c"]
Vertex "e" is adjacent to ["e", "a", "d"]
```

Adjacency Matrices

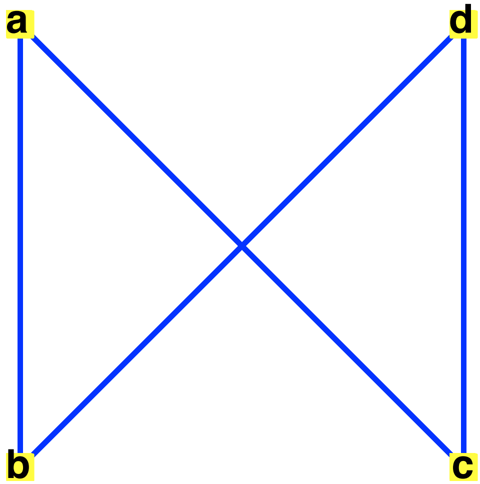
The adjacency matrix of a graph G with n vertices is the $n \times n$ matrix whose (i, j) entry is 1 if there is an edge from vertex i to vertex j and 0 if not. As with adjacency lists, you can define a graph by passing an adjacency matrix to the **Graph** command.

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

```
> exAdjM := Matrix([[0,1,1,0],[1,0,0,1],[1,0,0,1],[0,1,1,0]]);
```

$$exAdjM := \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \tag{10.60}$$

```
> exAdjMGraph := Graph(["a","b","c","d"],exAdjM):
> DrawGraph(exAdjMGraph);
```



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

Maple also provides a command, `AdjacencyMatrix`, for computing the adjacency matrix of a simple graph.

```
> AdjacencyMatrix(SpecialGraphs[WheelGraph](7));
```

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{10.61}$$

Incidence Matrices

The third representation of graphs we are considering are 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.

Unlike the other representations, Maple does not provide support for creating a graph from an incidence matrix. We will write a procedure to do so, at least for simple matrices.

To write this procedure, recall that the columns of the incidence matrix correspond to the edges of the graph. So we will use the columns to produce the list of edges. For each column, check each entry and add the row index to a set representing the edge. Assuming the incidence matrix is properly formed, each column will have only two 1s so each column will produce a two-element set representing an edge. The set of all of these forms the set of edges, which we can pass to the `Graph` command.

```
> GraphFromIncidence := proc(M::Matrix)
    local G, r, c, e, E;
    E := {};
    for c from 1 to LinearAlgebra[ColumnDimension](M) do
      e := {};
      for r from 1 to LinearAlgebra[RowDimension](M) do
        if M[r,c] = 1 then
          e := e union {r};
        end if;
      end do;
      E := E union {e};
    end do;
    G := GraphTheory[Graph](E);
  end proc:
```

As an example of our procedure, we reverse Example 6 from Section 10.3 and use the incidence matrix given in the solution in order to reproduce the graph.

```
> exIncMatrix := Matrix([[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]]);
```

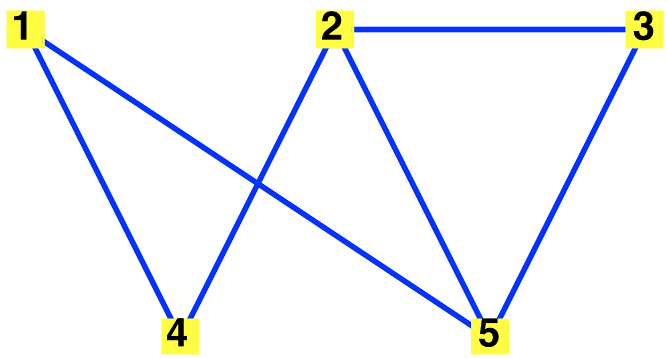
$$exIncMatrix := \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{10.62}$$

```
> exIncMGraph := GraphFromIncidence(exIncMatrix);
```

exIncMGraph := Graph 64: an undirected unweighted graph with 5 vertices and 6 edge(s) **(10.63)**

```
> SetVertexPositions(exIncMGraph,[[0,1],[1,1],[2,1],[.5,0],
  [1.5,0]]);
```

```
> DrawGraph(exIncMGraph);
```



On the other hand, Maple does provide a command for computing the incidence matrix for a graph: **`IncidenceMatrix`**.

```
> IncidenceMatrix(exIncMGraph);
```

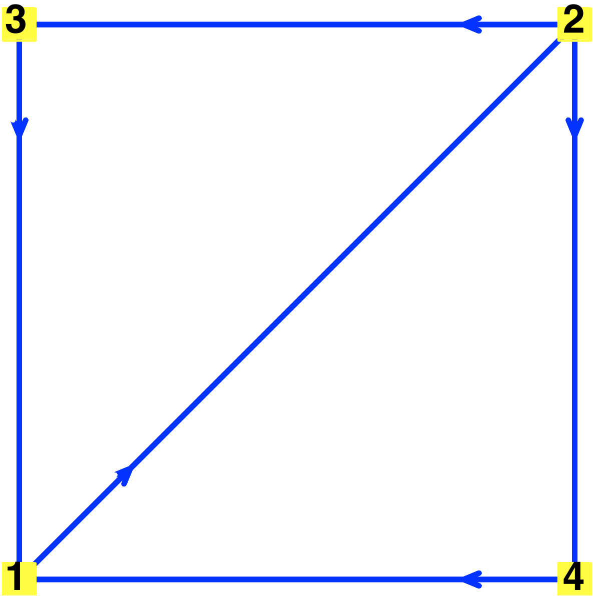
$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{10.64}$$

For a directed graph, the **`IncidenceMatrix`** command 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.

```
> directedIncidence := Digraph(Trail(1,2,3,1),Trail(2,4,1));
```

directedIncidence := Graph 65: a directed unweighted graph with 4 vertices and 5 arc(s) **(10.65)**

```
> SetVertexPositions(directedIncidence,[[0,0],[1,1],[0,1],[1,0]
  ]);
> DrawGraph(directedIncidence);
```



```
> IncidenceMatrix(directedIncidence);
```

$$\begin{bmatrix} 1 & 0 & 0 & -1 & -1 \\ -1 & 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{bmatrix} \tag{10.66}$$

Isomorphism of Graphs

We conclude this chapter 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 now create a procedure that will check some of the basic invariants that we've seen in this chapter: 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 Maple command `DegreeSequence` returns 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 Maple stores the vertices, it is not an invariant. However, applying the `sort` command to the result of the `DegreeSequence` command returns the degree sequence for the graph, as we defined it, which is an invariant.

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

```
> CheckInvariants := proc(G1::Graph, G2::Graph)
    local notIso;
    uses GraphTheory;
    notIso := false;
    if not (nops(Vertices(G1)) = nops(Vertices(G2))) then
      notIso := true;
      print("Different numbers of vertices");
    end if;
    if not (nops(Edges(G1)) = nops(Edges(G2))) then
      notIso := true;
      print("Different numbers of edges");
    end if;
    if IsDirected(G1) <> IsDirected(G2) then
      notIso := true;
      print("One is directed, one is undirected");
    end if;
    if IsBipartite(G1) <> IsBipartite(G2) then
      notIso := true;
      print("One is bipartite and the other is not");
    end if;
    if sort(DegreeSequence(G1))<>sort(DegreeSequence(G2)) then
      notIso := true;
      print("Degree sequences do not match");
    end if;
    if notIso then
      print("The graphs are not isomorphic");
    else
      print("The graphs MAY be isomorphic");
    end if;
  end proc:
```

```
> CheckInvariants(directedIncidence,exIncMGraph);
                              "Different numbers of vertices"
                               "Different numbers of edges"
                            "One is directed, one is undirected"
                              "Degree sequences do not match"
                              "The graphs are not isomorphic"                 (10.67)
```

```
> CheckInvariants(SpecialGraphs[HypercubeGraph](3),
  SpecialGraphs[PrismGraph](4));
                              "The graphs MAY be isomorphic"                  (10.68)
```

Maple provides a command, `IsIsomorphic`, for definitively determining whether or not two graphs are isomorphic. This command applies only to undirected and unweighted graphs. So, in particular, it can only be used on simple graphs, and not with any of the pseudographs we've created using edge weights and vertex attributes. The `IsIsomorphic` command accepts three arguments. The first two arguments are the two graphs to be compared. The third, optional, argument is a variable name in which the isomorphism, if it exists, is to be stored.

```
> IsIsomorphic(SpecialGraphs[HypercubeGraph](3),SpecialGraphs
  [PrismGraph](4),'hyperprismiso');
                                        true                                  (10.69)
```

```
> hyperprismiso;
                                                                              (10.70)
```

["000" = 1, "001" = 2, "010" = 4, "011" = 3, "100" = 5, "101" = 6, "110" = 8, "111" = 7] **(10.70)**

Isomorphic Pseudographs

The `IsIsomorphic` command applies only to simple graphs. We now present a procedure to determine if two pseudographs (undirected but allowing loops and multiple edges) are isomorphic.

```
> PseudoIsomorphic := proc(G1::Graph, G2::Graph)
    local isoFound, V1, V2, P, permindex, i, j, n;
    uses GraphTheory;
    isoFound := false;
    V1 := Vertices(G1);
    n := nops(V1);
    if (n <> nops(Vertices(G2))) then
      print("Graphs have differing numbers of vertices.");
      return false;
    end if;
    P := combinat[permute](Vertices(G2));
    permindex := 1;
    while (not isoFound) and (permindex <= n!) do
      isoFound := true;
      V2 := P[permindex];
      for i from 1 to n do
        if Degree(G1,V1[i]) <> Degree(G2,V2[i]) then
          isoFound := false;
          next;
        end if;
      end do;
      for i from 1 to n do
        if (GetVertexAttribute(G1,V1[i],"loop") and
              (not GetVertexAttribute(G2,V2[i],"loop")))
              or ((not GetVertexAttribute(G1,V1[i],"loop"))
                and GetVertexAttribute(G2,V2[i],"loop")) then
          isoFound := false;
          next;
        end if;
      end do;
      for i from 1 to n do
        for j from 1 to n do
          if i = j then next; end if;
          if HasEdge(G1,{V1[i],V1[j]})
              xor HasEdge(G2,{V2[i],V2[j]}) then
            isoFound := false;
            break;
          elif IsWeighted(G1)
               and HasEdge(G1,{V1[i],V1[j]}) then
            if GetEdgeWeight(G1,{V1[i],V1[j]})
                 <> GetEdgeWeight(G2,{V2[i],V2[j]}) then
              isoFound := false;
              break;
            end if;
          end if;
        end do;
        if not isoFound then
          break;
        end if;
      end do;
      permindex := permindex + 1;
    end do;
    if isoFound then
      printf("Found an isomorphism\n");
      for i from 1 to n do
        printf("%a -> %a\n",V1[i],V2[i]);
      end do;
    else
      printf("There is no isomorphism");
    end if;
  end proc:
```

The basic idea of this procedure is that a mapping from the graph **G1** to the graph **G2** is represented by a permutation of the vertices of **G2**. More specifically, the variable **V1** is set to the list of the vertices of the graph **G1**, and does not change. The variable **V2** does change: for each iteration of the main while loop, **V2** is set to a different permutation of the vertices of the graph **G2**. This represents the map that sends the vertex in the *i*th position of **V1** to the vertex in the *i*th position in **V2**. The bulk of the procedure is concerned with checking to see if this map is an isomorphism.

Other local variables used in the procedure are **n**, **i**, **j**, **permindex**, and **P**. The variable **n** is the number of vertices in **G1**, which the procedure confirms is the same as the number in **G2**. The variables **i** and **j** are indices used when checking the edges of the graphs. **P** is the list of all permutations of the vertices of **G2** and **permindex** is a counter used to reference which permutation is under consideration.

The last variable, **isoFound**, is a boolean variable used to end the while loop if an isomorphism is found. The **isoFound** variable is initialized to false — since the while loop includes the **not isoFound** condition, a value of false for the variable allows the while loop to continue. Inside the while loop, **isoFound** is immediately set to true, indicating the assumption that the current permutation represents an isomorphism. If one of the tests inside the while loop determines that it is not an isomorphism, it sets **isoFound** back to false, allowing the while loop to continue. If none of the tests do so, then **isoFound** remains true, the while loop is exited, and the procedure displays the isomorphism. The other possible reason to exit the while loop is that the permutations have been exhausted, in which case the procedure reports that the graphs are not isomorphic.

We also use the **next** and **break** commands. Inside of a loop, a **next** statement causes Maple to skip the rest of the statements in the do block and move on to the next iteration of the loop. If there is an index of iteration, as in a for loop, that index is incremented, and the termination conditions are checked. The **next** command is a useful way to avoid unnecessary computation and slightly improve performance. In this case, once we know that the permutation is not an isomorphism, it's not necessary to continue with the tests. The **break** command is similar, but instead of moving to the next iteration of the loop, **break** causes the loop to immediately terminate. We use **break** when testing the edges to avoid continuing to test after finding an incompatible pair.

Within the while loop, the **PseudoIsomorphic** procedure performs three tests. First, it makes sure that the degrees of corresponding vertices are the same. If any are different, it immediately knows that the permutation cannot represent an isomorphism, saving it from more computationally expensive tests.

Second, it tests to ensure that corresponding vertices are both looped or both not looped. If one is marked as having a loop and the other is not, then the permutation is not an isomorphism.

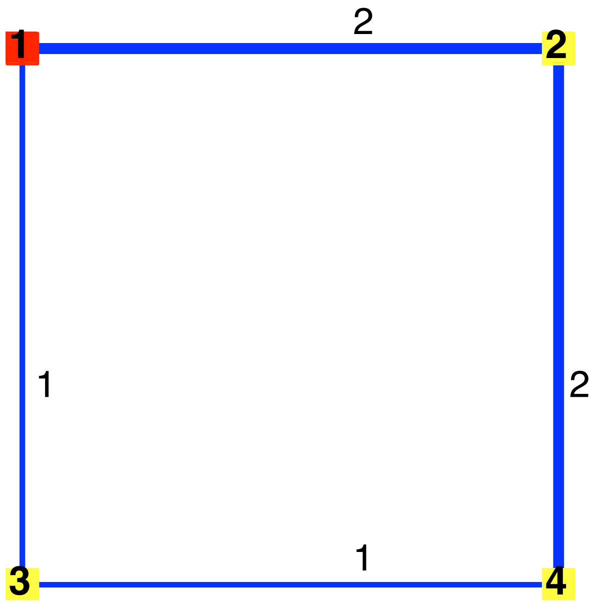
Finally, the procedure tests to make sure that the graphs have the same edges. The variables **i** and `j` represent indices to vertices, so that the double for loops are looping through every pair of vertices. The `xor` test checks to see if the result of `HasEdge` on corresponding vertices returns different values. That is, if one graph has the specified edge and the other does not, then one of the `HasEdge` statements will return true and the other false, resulting in their xor to be true. In this case, the graphs do not coincide. The second part of the test applies only to weighted graphs that have the edge in question. In that case, the procedure checks to see that the weights of the edges are the same, indicating that the multiple edges coincide.

As a first example, we check that this procedure agrees with `IsIsomorphic` on a pair of simple graphs.

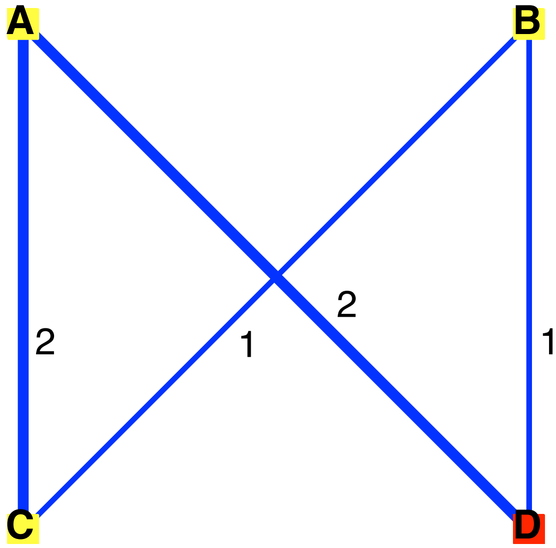
```
> PseudoIsomorphic(SpecialGraphs[HypercubeGraph](3),
  SpecialGraphs[PrismGraph](4));
Found an isomorphism
"000" -> 1
"001" -> 2
"010" -> 4
"011" -> 3
"100" -> 5
"101" -> 6
"110" -> 8
"111" -> 7
```

Here is a pair of pseudographs. The procedure produces an isomorphism between them.

```
> IsoTest1 := Graph({[{1,2},2],{1,3},[{2,4},2],{3,4}}):
> SetVertexAttribute(IsoTest1,1,"loop"=true);
> IsoTest2 := Graph({[{"A","C"},2],[{"A","D"},2],{"B","C"},
  {"B","D"}}):
> SetVertexAttribute(IsoTest2,"D","loop"=true);
> SetVertexPositions(IsoTest1,[[0,1],[1,1],[0,0],[1,0]]);
> SetVertexPositions(IsoTest2,[[0,1],[1,1],[0,0],[1,0]]);
> DrawPseudograph(IsoTest1);
```



```
> DrawPseudograph(IsoTest2);
```



```
> PseudoIsomorphic(IsoTest1,IsoTest2);
Found an isomorphism
1 -> "C"
2 -> "A"
3 -> "B"
4 -> "D"
```

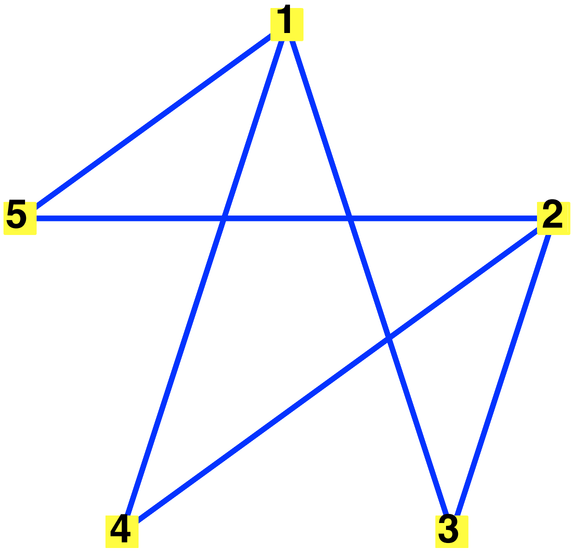
10.4 Connectivity

Maple provides a number of commands related to connectivity of graphs.

Connectedness in Undirected Graphs

The first such command that we consider is the **IsConnected** command. This command 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.

```
> DrawGraph(CompleteGraph(2,3),style=circle);
```

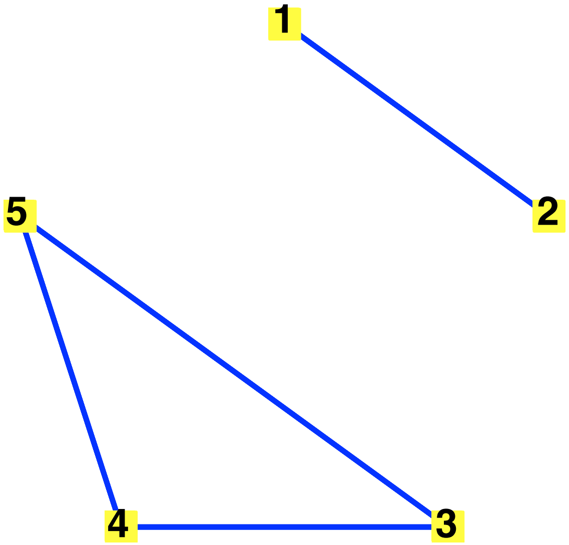


```
> IsConnected(CompleteGraph(2,3));
```

(10.71)

```
                              true                              (10.71)
```

```
> DrawGraph(GraphComplement(CompleteGraph(2,3)),style=circle);
```



```
> IsConnected(GraphComplement(CompleteGraph(2,3)));
                              false                             (10.72)
```

Connectivity and Vertices

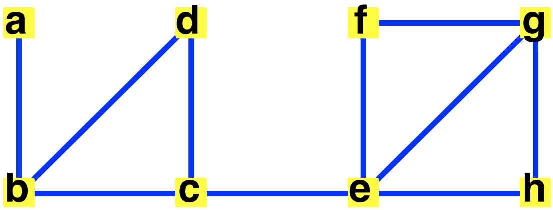
In addition to testing whether a graph is connected or not, Maple also has commands for determining which vertices are cut vertices (which Maple refers to as articulation points) and for calculating both the vertex and edge connectivity.

First, let us recreate two of the examples from Figure 4 of Section 10.4, namely G_1 and G_3.

```
> Figure4G1 := Graph({{"a","b"},{"b","c"},{"b","d"},{"c","d"},
  {"c","e"},{"e","f"},{"e","g"},{"e","h"},{"f","g"},{"g","h"}})
  ;
Figure4G1 := Graph 66: an undirected unweighted graph with 8 vertices and 10 edge(s)  (10.73)
```

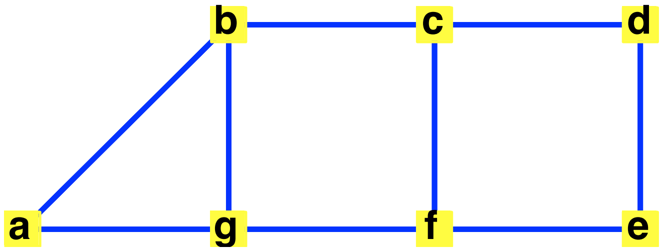
```
> SetVertexPositions(Figure4G1,[[0,1],[0,0],[1,0],[1,1],[2,0],
  [2,1],[3,1],[3,0]]);
```

```
> DrawGraph(Figure4G1);
```



```
> Figure4G3 := Graph({{"a","b"},{"a","g"},{"b","c"},{"b","g"},
  {"c","d"},{"c","f"},{"d","e"},{"e","f"},{"f","g"}});
Figure4G3 := Graph 67: an undirected unweighted graph with 7 vertices and 9 edge(s)   (10.74)
```

```
> SetVertexPositions(Figure4G3,[[0,0],[1,1],[2,1],[3,1],[3,0],
  [2,0],[1,0]]);
> DrawGraph(Figure4G3);
```



To find the cut vertices (or articulation points), we use the command `ArticulationPoints`. This command takes only one argument, the name of a graph, and returns a list of vertices that are articulation points, *i.e.*, vertices which, if removed, would disconnect the graph.

```
> ArticulationPoints(Figure4G1);
                              ["b", "c", "e"]                          (10.75)
> ArticulationPoints(Figure4G3);
                                    [ ]                                (10.76)
```

These results indicate that G_1 has three cut vertices while G_3 has none.

In other words, G_3 is nonseparable. Recall that a nonseparable graph will have vertex connectivity $\kappa(G) \geq 2$ and is thus referred to as 2-connected or biconnected, provided it has at least 3 vertices. The Maple command `IsBiconnected` also indicates that `Figure4G3` is nonseparable.

```
> IsBiconnected(Figure4G3);
                                   true                                (10.77)
```

Finally, the `VertexConnectivity` command computes $\kappa(G)$, the minimum number of vertices that must be deleted in order to disconnect a graph. The only argument is the name of the graph.

```
> VertexConnectivity(Figure4G1);
                                     1                                 (10.78)
> VertexConnectivity(Figure4G3);
                                     2                                 (10.79)
```

Finding a vertex cut

We conclude our discussion of vertex connectivity by developing a procedure for determining which sets of vertices in a graph G form a vertex cut of size $\kappa(G)$. That is, we we want to find a minimal set of vertices which separate the graph. Maple tells us how many vertices are in such a set, but does not have a command for finding them.

First, we create a command that will test whether or not a given set of vertices is or is not a vertex cut. We do this by simply removing the vertices from the graph with the `DeleteVertex` command and then testing the resulting graph for connectedness with `IsConnected`.

```
> IsVertexCut := proc(G::Graph, V::list)
    local H, iscut;
    uses GraphTheory;
    H := DeleteVertex(G,V);
    iscut := not IsConnected(H);
    return iscut;
  end proc:
```

We see that, in G_3, $\{c, f\}$ separates the graph, while $\{a, d\}$ does not.

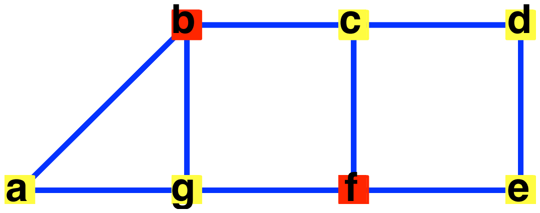
```
> IsVertexCut(Figure4G3,["c","f"]);
                              true                              (10.80)
> IsVertexCut(Figure4G3,["a","d"]);
                              false                             (10.81)
```

Now we write the procedure to find all minimal vertex cuts. We will do this by brute force. First, the `Vertices` command produces the list of vertices in the graph. Then the `choose` command from the `combinat` package takes the list of vertices and the value of $\kappa(G)$ and produces a list of all sublists of vertices of that size. Then we check each of those sublists with `IsVertexCut` to see which are vertex cuts.

```
> FindVertexCuts := proc(G::Graph)
    local k, subLists, testVerts, result;
    uses GraphTheory;
    k := VertexConnectivity(G);
    subLists := combinat[choose](Vertices(G),k);
    result := [];
    for testVerts in subLists do
      if IsVertexCut(G,testVerts) then
        result := [op(result),testVerts];
      end if;
    end do;
    return result;
  end proc:
```

We apply this to G_3 to find the possible minimum vertex cuts and then we redraw the graph with one of the choices highlighted red.

```
> Figure4G3VCs := FindVertexCuts(Figure4G3);
 Figure4G3VCs := [["b", "f"], ["b", "g"], ["c", "e"], ["c", "f"], ["c", "g"], ["d", "f"]]   (10.82)
> HighlightVertex(Figure4G3,Figure4G3VCs[1],red);
> DrawGraph(Figure4G3);
```



Connectivity and Edges

Maple offers similar functionality for edges. Unlike for vertices, however, Maple does not include a command like `ArticulationPoints` that will list all of a graph's bridges (recall that a bridge or a cut edge is an edge whose removal will disconnect the graph). Maple does, however, have a command to test whether an particular edge is a bridge.

The `IsCutSet` command takes two arguments. The first is the name of the graph. The second argument can be a single edge, in which case the function determines whether that edge is a bridge or not. Alternately, the second argument may be a set of edges, in which case the function determines whether or not that set is an edge cut. For example, we see that edge $\{c, e\}$ in G_1 is a bridge.

```
> IsCutSet(Figure4G1,{"c","e"});
                              true                              (10.83)
```

We can also see that the pair of edges $\{b, c\}$ and $\{g, f\}$ form an edge cut of G_3.

```
> IsCutSet(Figure4G3,{{"b","c"},{"g","f"}});
                              true                              (10.84)
```

We now use this to create a command analogous to `ArticulationPoints`. This procedure works by checking each edge to see if its removal disconnects the graph. Remember that, unlike `DeleteVertex`, the default behavior of `DeleteEdge` is to replace the original graph with the graph with the given edge removed. We override this default behavior by using the option `inplace=false`.

```
> Bridges := proc(G::Graph)
    local H, edges, E, bridges;
    uses GraphTheory;
    edges := Edges(G);
    bridges := [];
    for E in edges do
      H := DeleteEdge(G,E,inplace=false);
      if not IsConnected(H) then
        bridges := [op(bridges),E];
      end if;
    end do;
    return bridges;
  end proc:
```

We can use this to see that G_1 has two bridges and that G_3 has none.

```
> Bridges(Figure4G1);
                    [{"a", "b"}, {"c", "e"}]                    (10.85)
> Bridges(Figure4G3);
                              [ ]                               (10.86)
```

Finally, Maple will compute $\lambda(G)$, the edge connectivity of the graph, with the command `EdgeConnectivity`. Just like `VertexConnectivity`, the only argument that is accepted is the name of the graph, and the command returns the minimum number of edges that must be deleted in order to disconnect the graph.

We have already seen that G_1 has bridges, and thus has edge connectivity 1.

```
> EdgeConnectivity(Figure4G1);
                               1                                (10.87)
```

On the other hand, the `Bridges` command verified that G_3 does not have bridges, but it does have an edge cut of size 2.

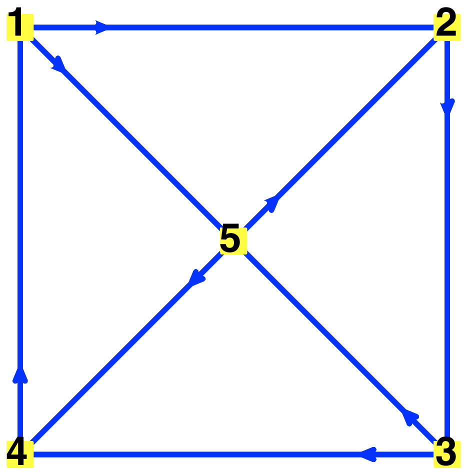
```
> EdgeConnectivity(Figure4G3);
                               2                                (10.88)
```

Connectedness in Directed Graphs

When used with a directed graph, `IsConnected` returns true if the underlying undirected graph is connected. That is, for directed graphs, `IsConnected` determines whether or not the graph is weakly connected. To check if a directed graph is strongly connected, Maple has the command `IsStronglyConnected`.

We consider a pair of examples.

```
> strongEx := Digraph(Trail(1,2,3,4,1),{[1,5],[5,2],[3,5],[5,4]
  }):
> SetVertexPositions(strongEx,[[0,1],[1,1],[1,0],[0,0],[0.5,
  0.5]]);
> DrawGraph(strongEx);
```



```
> IsConnected(strongEx);
```

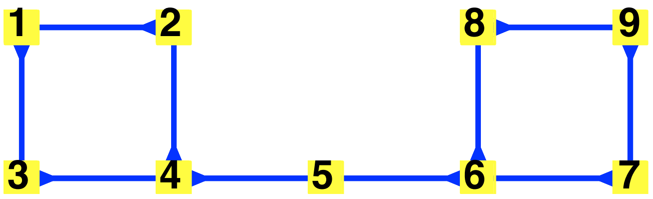
```
                              true                              (10.89)
> IsStronglyConnected(strongEx);
                              true                              (10.90)
```

Applying **IsConnected** and **IsStronglyConnected** indicates that this graph is strongly connected.

```
> IsConnected(strongEx);
                              true                              (10.91)
> IsStronglyConnected(strongEx);
                              true                              (10.92)
```

The second example we create will be seen to be weakly, but not strongly, connected.

```
> weakEx := Digraph(Trail(4,2,1,3,4,5),Trail(6,8,9,7,6,5));
    weakEx := Graph 68: a directed unweighted graph with 9 vertices and 10 arc(s)    (10.93)
> SetVertexPositions(weakEx,[[0,1],[1,1],[0,0],[1,0],[2,0],[3,
  0],[4,0],[3,1],[4,1]]);
> DrawGraph(weakEx);
```



```
> IsConnected(weakEx);
                              true                              (10.94)
> IsStronglyConnected(weakEx);
                              false                             (10.95)
```

Maple also has commands to extract the connected components of a graph that is not connected. The **ConnectedComponents** command takes a graph as input and returns a list of lists of vertices. For directed graphs, **ConnectedComponents** is used to determine the weakly connected components of the graph. The strongly connected components are obtained with **StronglyConnectedComponents**.

```
> ConnectedComponents(GraphComplement(CompleteGraph(2,3)));
                        [[1, 2], [3, 4, 5]]                     (10.96)
```

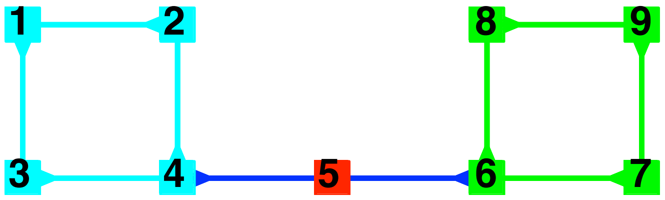
The above indicates that the complement of $K_{2,\,3}$ has two connected components. The first component consists of the subgraph comprised of vertices 1 and 2, and the second connected component consists of the other three vertices.

Coloring the Components

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

```
> HighlightSCC := proc(G::Graph)
    local colorList, components, c, i, H;
    colorList := [red,green,cyan,brown,gray];
    components := StronglyConnectedComponents(G);
    c := 0;
    for i from 1 to nops(components) do
      c := c + 1;
      if c > 5 then c := 1; end if;
      if nops(components[i]) = 1 then
        HighlightVertex(G,components[i],colorList[c]);
      else
        H := InducedSubgraph(G,components[i]);
        HighlightSubgraph(G,H,colorList[c],colorList[c]);
      end if;
    end do;
    DrawGraph(G);
  end proc:
```

```
> HighlightSCC(weakEx);
```



Counting Paths Between Vertices

The last topic that we will 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 `strongEx` graph from above. We can obtain its adjacency matrix by applying the `AdjacencyMatrix` command to the name of the graph.

```
> Amatrix := AdjacencyMatrix(strongEx);
```

$$Amatrix := \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{10.97}$$

Next, compute some powers of the adjacency matrix.

```
> Amatrix^2, Amatrix^3, Amatrix^4;
```

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 0 & 2 & 2 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 2 & 1 & 2 & 2 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 & 0 \end{bmatrix} \tag{10.98}$$

```
> Amatrix^5, Amatrix^6, Amatrix^7;
```

$$\begin{bmatrix} 2 & 3 & 2 & 2 & 1 \\ 0 & 2 & 1 & 2 & 2 \\ 2 & 2 & 2 & 3 & 1 \\ 1 & 2 & 0 & 2 & 2 \\ 2 & 1 & 2 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 2 & 3 & 3 & 3 & 4 \\ 2 & 2 & 2 & 3 & 1 \\ 3 & 3 & 2 & 3 & 4 \\ 2 & 3 & 2 & 2 & 1 \\ 1 & 4 & 1 & 4 & 4 \end{bmatrix}, \begin{bmatrix} 3 & 6 & 3 & 7 & 5 \\ 3 & 3 & 2 & 3 & 4 \\ 3 & 7 & 3 & 6 & 5 \\ 2 & 3 & 3 & 3 & 4 \\ 4 & 5 & 4 & 5 & 2 \end{bmatrix} \tag{10.99}$$

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 Maple 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 Maple uses the word cycle in its help pages. These two terms are synonymous.)

Euler Circuits in Simple Graphs

Maple comes equipped with a command to determine if a given simple graph has n Euler circuit or not. This command, `IsEulerian`, takes one or two arguments. The required argument is the graph. The second argument is an optional name in which Maple will store the Eulerian circuit it finds. As an example, we'll have Maple find an Euler circuit on K_5.

```
> IsEulerian(CompleteGraph(5), 'K5EulerCircuit');
```

$$true \tag{10.100}$$

```
> K5EulerCircuit;
```

(10.101)

$$Trail(1, 2, 3, 1, 4, 2, 5, 3, 4, 5, 1) \tag{10.101}$$

Now we'll have Maple help us visualize this path by creating an animation that successively highlights the edges in the path. To do this we will use the **animate** command. The **animate** command takes at least three arguments. The first is a Maple procedure that generates a plot. Typically, one uses one of the built-in commands, such as **plot** or **plot3d** as the first argument. In this case, however, we will create our own procedure, `plotPath`, for the first argument. We will return to `plotPath` in a moment. The second argument to the **animate** command is a list containing the arguments to the command given in the first argument. The third argument will be a parameter with a range specification of the form `t=a..b` which specifies the parameter used in the construction of the individual plots that make up the animation and their bounds. We will also be using two options. The `paraminfo=false` option turns off the display of the value of the parameter, while `frames=50` tells Maple to create 50 frames instead of the default 25, which in this case has the effect of slowing down the animation. (Note: there is no way to change the frame rate of the animation with the command line, but you can increase and decrease the frames per second (FPS) in the context menu of an animation. You can also step through the animation one frame at a time to better see the progress of the path.)

We now return to the `plotPath` procedure, which will be the first argument to the **animate** command. The `plotPath` procedure will take as arguments a graph, a list of vertices representing a path, and a number representing the progress along the path (*e.g.*, 1 indicates one edge traversed, 2 indicates two edges traverse, *etc.*).

It first makes a copy of the graph so that the modifications to the edge colors do not affect the original graph. The procedure uses the local variable `N` to ensure that it does not exceed the length of the list given in the second parameter and to ensure that the value representing the progress along the path is an integer. Assuming the requested path length is not 0, then the procedure takes a slice out of the list of vertices to represent a path of that length and highlights that "trail." In the case that the third argument is 0, it skips the highlighting steps and just draws the graph.

```
> plotPath := proc(G::Graph, P::list, n)
    local Gcopy, path, N;
    uses GraphTheory;
    Gcopy := CopyGraph(G);
    if n > nops(P) - 1 then
      N := nops(P) - 1;
    else
      N := floor(n);
    end if;
    if N <> 0 then
      path := P[1..(N+1)];
      HighlightTrail(Gcopy,path);
    end if;
    DrawGraph(Gcopy);
  end proc:
```

Now we use the `plotPath` procedure as the basis for the following `animatePath` procedure. This procedure will take as input a graph and a path and create the animation using the **animate** command.

```
> animatePath := proc(G::Graph, P::list)
    local t;
    plots[animate](plotPath, [G,P,t], t=0..(nops(P)-1),
                    paraminfo=false, frames=50);
```

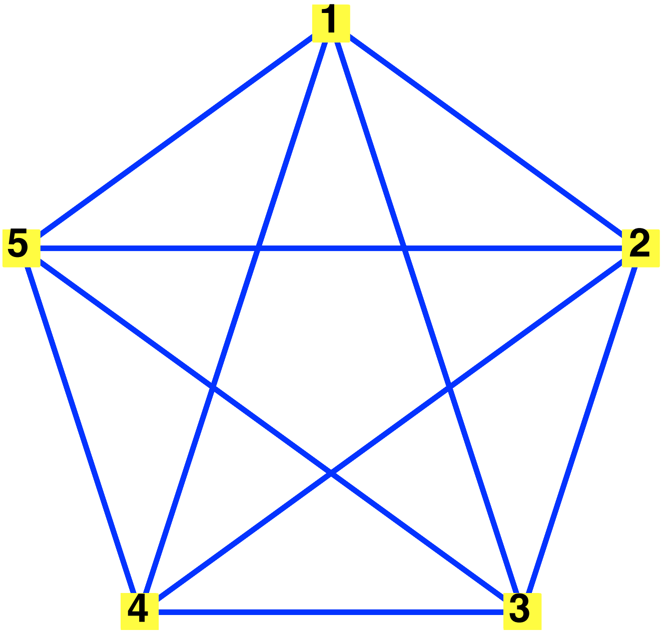
```
end proc:
```

To use this procedure, we just need to turn the "trail" that **IsEulerian** found above into a list of vertices.

```
> K5CircuitList := [op(K5EulerCircuit)];
```

$$K5CircuitList := [1, 2, 3, 1, 4, 2, 5, 3, 4, 5, 1] \quad \textbf{(10.102)}$$

```
> animatePath(CompleteGraph(5),K5CircuitList);
```



You can see the Euler circuit traced out by clicking on the image of the graph and then clicking on the play button at the top of the window.

Euler Circuits in Multigraphs

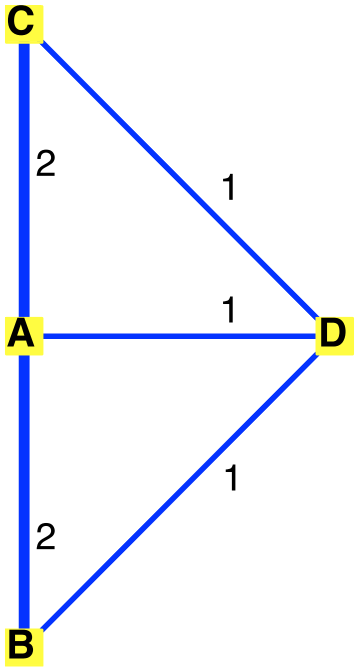
As usual, Maple's built-in function only applies to simple graphs, *i.e.*, graphs with no loops or multiple edges. We will examine the problem of finding Euler circuits in multigraphs. We know, from Theorem 1 of Section 10.5, that a connected multigraph with at least two vertices has an Euler circuit 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 procedure for determining whether or not a pseudograph has an Euler circuit.

```
> IsPseudoEulerian := proc(G::Graph)
    local v;
    uses GraphTheory;
    if IsDirected(G) then
      return FAIL;
    end if;
    if (not IsConnected(G)) or (nops(Vertices(G)) < 2) then
      return false;
    end if;
    for v in Vertices(G) do
      if type(PseudoDegree(G,v),odd) then
        return false;
      end if;
    end do;
    return true;
```

```
  end proc:
```

We can use this procedure 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.

```
> Konigsberg := Graph({[{"A","B"},2], [{"A","C"},2],
                       {"A","D"}, {"B","D"}, {"C","D"}});
 Konigsberg := Graph 69: an undirected weighted graph with 4 vertices and 5 edge(s)   (10.103)
> SetVertexPositions(Konigsberg,[[0,1],[0,0],[0,2],[1,1]]);
> DrawPseudograph(Konigsberg);
```



```
> IsPseudoEulerian(Konigsberg);
                                false                                        (10.104)
```

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.

```
> FindMultiEuler := proc(G::Graph)
    local H, circuit, subC, i, v, insertPoint, e, w,
          buildingSub, oldC;
    uses GraphTheory;
    if not IsPseudoEulerian(G) then
      return false;
    end if;
    H := CopyGraph(G);
    circuit := [];
    while Edges(H) <> {} do
      # find a starting point
      if circuit = [] then
        subC := [Vertices(H)[1]];
      else
        for i from 1 to nops(circuit) do
```

```
          if Neighbors(H,circuit[i]) <> [] then
            subC := [circuit[i]];
            insertPoint := i;
            break;
          end if;
        end do;
      end if;
      # build a subcircuit
      buildingSub := true;
      while buildingSub do
        v := subC[-1];
        w := Neighbors(H,v)[1];
        e := {v,w};
        if IsWeighted(H) then
          if GetEdgeWeight(H,e) > 1 then
            SetEdgeWeight(H,e,GetEdgeWeight(H,e)-1);
          else
            DeleteEdge(H,e);
          end if;
        else
          DeleteEdge(H,e);
        end if;
        subC := [op(subC),w];
        if w = subC[1] then
          buildingSub := false;
        end if;
      end do;
      # splice the subcircuit into the main circuit
      if circuit = [] then
        circuit := subC;
      else
        oldC := circuit;
        circuit := [];
        if insertPoint >= 2 then
          circuit := oldC[1..(insertPoint-1)];
        end if;
        circuit := [op(circuit),op(subC)];
        if insertPoint < nops(oldC) then
          circuit:=[op(circuit),op(oldC[(insertPoint+1)..-1])];
        end if;
      end if;
    end do;
    return circuit;
  end proc:
```

The program begins with a use of `IsPseudoEulerian` in order to avoid searching for a circuit that cannot exist. It then assigns to the variable `H` a copy of the graph. It is this copy that is used throughout the rest of the procedure, rather than the graph `G` that was passed to the algorithm. The benefit of using a copy is that the procedure 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 variable `circuit` will hold the main circuit that, at the end of the procedure, is output to the user. The circuit will be stored as a list of the vertices 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.*, it is the first pass through the main loop), then the starting point is the first vertex 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` variable 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`. The variable `e = {v,w}` is therefore an edge in the graph that has not already been traversed by the circuit. The nested if statements that follow the assignment of e effect the removal of the edge `e` from the graph `H`. In the case that `H` is weighted (*i.e.*, is a multigraph), the weight is either decreased by 1 to represent the removal of one of several edges between the vertices or is deleted if there is only one such edge. In the unweighted case, the edge is always deleted from the graph. After the edge has been deleted, the vertex `w` is added to the subcircuit, representing the inclusion of the edge. Finally, 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 while 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. In the first pass through the main loop, the main circuit is empty and so `subC` is just copied into `circuit`. In subsequent passes of the main loop, the variable `oldC` is used to store the "old" circuit. Recall that `insertPoint` stores the index of the starting vertex for `subC`. The goal is to put the subcircuit in that location. The new, more complete, circuit is built in three pieces. First, the part of the old circuit that occurs before the insertion point (assuming the insertion point is not the initial vertex of the main circuit). Second, the subcircuit. And third, the part of the old circuit that comes after the insertion point (assuming the insertion point is not the final vertex).

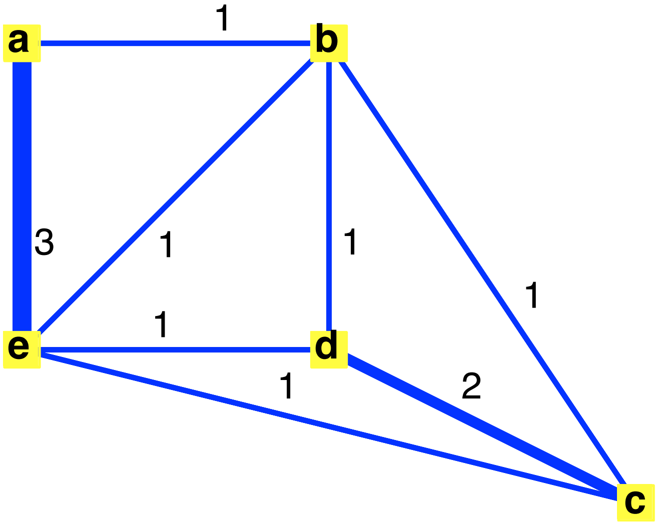
The main while loop continues until all the edges of the graph have been included in the circuit, making `circuit` an Euler circuit for the graph. As an example, consider Exercise 5 from Section 10.5.

```
> Ex5 := Graph({{"a","b"},[{"a","e"},3],{"b","c"},{"b","d"},
  {"b","e"},[{"c","d"},2],{"c","e"},{"d","e"}});
```

Ex5 := Graph 70: an undirected weighted graph with 5 vertices and 8 edge(s) **(10.105)**

```
> SetVertexPositions(Ex5,[[0,2],[1,2],[2,.5],[1,1],[0,1]]);
```

```
> DrawPseudograph(Ex5);
```



```
> Ex5Path := FindMultiEuler(Ex5);
```

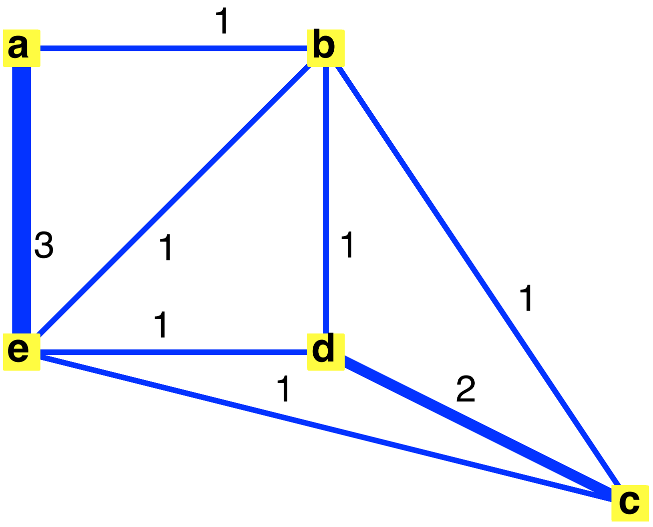
$Ex5Path := ["a", "e", "c", "d", "e", "a", "b", "c", "d", "b", "e", "a"]$ **(10.106)**

Note that the edge between *a* and *e* is traversed three times: as the first edge in the circuit, shortly before the middle of the circuit, and again as the last edge in the circuit. This is consistent with there being three edges between *a* and *e*.

The following procedures can be used to create animations to visualize the circuit in an multigraph, as `animatePath` did above for simple graphs. `HighlightMultiTrail` replaces `HighlightTrail` and serves to transition multi-edges from blue to red in steps. Note the use of edge attributes to track the number of times a multi-edge has been traversed.

```
> HighlightMultiTrail := proc(G::Graph, T)
    local H, i, e, x, redshade;
    uses GraphTheory;
    if not IsWeighted(G) then
      H:= MakeWeighted(G);
    else
      H := CopyGraph(G);
    end if;
    for e in Edges(H) do
      SetEdgeAttribute(H,e,"traversed"=0);
    end do;
    for i from 2 to nops(T) do
      e := {T[i-1],T[i]};
      x := GetEdgeAttribute(H,e,"traversed")+1;
      SetEdgeAttribute(H,e,"traversed"=x);
    end do;
    for e in Edges(G) do
      redshade := GetEdgeAttribute(H,e,"traversed")
                                      /GetEdgeWeight(H,e);
      HighlightEdges(G,{e},COLOR(RGB,redshade,0,1-redshade));
    end do;
    return G;
  end proc:
```

```
> PlotMultiPath := proc(G::Graph, P::list, n)
    local Gcopy, path, N;
    uses GraphTheory;
    Gcopy := CopyGraph(G);
    if n > nops(P) - 1 then
      N := nops(P) - 1;
    else
      N := floor(n);
    end if;
    if N <> 0 then
      path := P[1..(N+1)];
      HighlightMultiTrail(Gcopy,path);
    end if;
    DrawGraph(Gcopy);
  end proc:
> AnimateMultiPath := proc(G::Graph, P::list)
    local t;
    plots[animate](PlotMultiPath, [G,P,t], t=0..(nops(P)-1),
                                 paraminfo=false, frames=50);
  end proc:
> AnimateMultiPath(Ex5,Ex5Path);
```

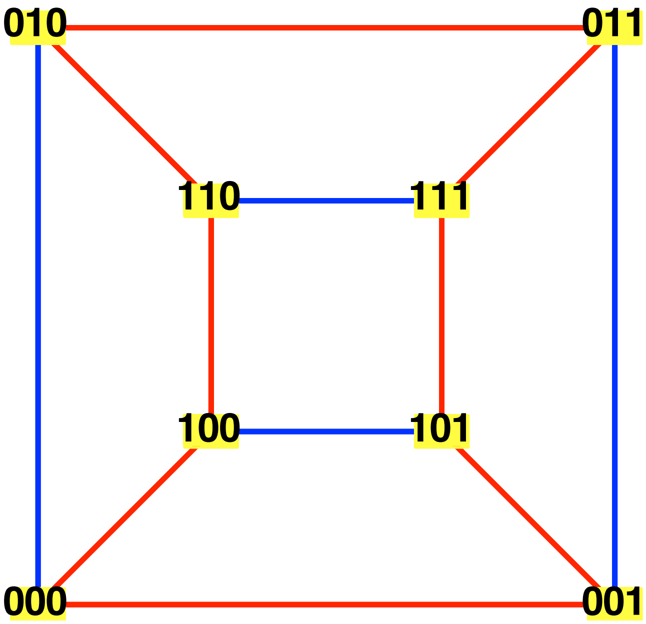


Hamilton Circuits

Turning our attention to Hamilton circuits, Maple provides the command **IsHamiltonian** for determining whether or not the graph contains a Hamilton circuit. This command, like **IsEulerian**, accepts one required and one optional argument. The required argument, of course, is a graph. If a variable name is provided as the second argument, then Maple will store the Hamilton circuit in the variable, which you can then use as the second argument to **HighlightTrail**.

```
> HCGraph := SpecialGraphs[HypercubeGraph](3);
 HCGraph := Graph 71: an undirected unweighted graph with 8 vertices and 12 edge(s) (10.107)
> IsHamiltonian(HCGraph,'HCpath');
                                  true                                            (10.108)
```

```
> HighlightTrail(HCGraph,HCpath);
> DrawGraph(HCGraph);
```



```
> IsHamiltonian(CompleteGraph(3,2));
                                  false                                   (10.109)
```

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 `IsHamiltonian` command to pseudographs.

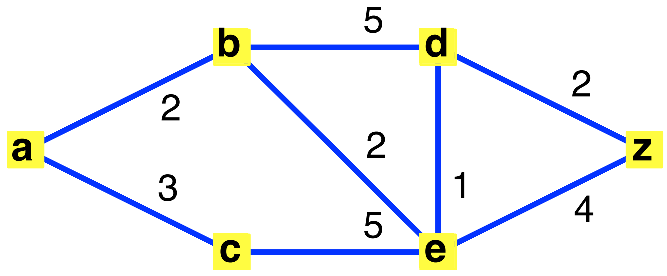
10.6 Shortest-Path Problems

Among the most common problems in graph theory are the "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.

In the previous sections of this chapter, we used edge weights as a way to get around Maple's limitation of being able to represent simple graphs only. In this section, we will use weighted graphs in the way they are actually intended — to represent some sort of cost associated with traversing the edge. Note that pseudographs are rarely, if ever, of use in shortest path problems. There is no reason to consider multiple edges between two vertices since the edge of lowest weight is always preferred. And traversing a loop at a vertex would only increase the cost with no benefit.

First, we reproduce Exercise 2 from Section 10.6 of the textbook to use as an example. Recall that when defining an undirected and weighted graph, we use the format `[{a,b},w]` to indicate that the graph has an edge between *a* and *b* with weight *w*.

```
> Ex2 := Graph({[{"a","b"},2],[{"a","c"},3],
                [{"b","d"},5],[{"b","e"},2],
                [{"c","e"},5],[{"d","e"},1],
                [{"d","z"},2],[{"e","z"},4]});
  Ex2 := Graph 72: an undirected weighted graph with 6 vertices and 8 edge(s)   (10.110)
> SetVertexPositions(Ex2,[[0,.5],[1,1],[1,0],[2,1],[2,0],[3,.5]
  ]);
> DrawGraph(Ex2);
```



Now we will make use of Maple's implementation of Dijkstra's algorithm to compute the shortest path between *a* and *z*. To do this, we simply call the **DijkstrasAlgorithm** command with three arguments: the graph and the names of the starting and ending vertices.

```
> DijkstrasAlgorithm(Ex2,"a","z");
                    [["a", "b", "e", "d", "z"], 7]                    (10.111)
```

The output informs us that the shortest path is *a*, *b*, *e*, *d*, *z* and that the length of this path is 7.

There is an alternate form of the command for producing the shortest path from an initial vertex to several different vertices at once.

```
> DijkstrasAlgorithm(Ex2,"a",["d","e","z"]);
   [[["a", "b", "e", "d"], 5], [["a", "b", "e"], 4], [["a", "b", "e", "d", "z"], 7]]   (10.112)
```

And for producing the shortest paths from the starting vertex to all other vertices.

```
> DijkstrasAlgorithm(Ex2,"a");
[[["a"], 0], [["a", "b"], 2], [["a", "c"], 3], [["a", "b", "e", "d"], 5], [["a", "b", "e"],   (10.113)
    4], [["a", "b", "e", "d", "z"], 7]]
```

To determine the shortest path from every vertex to every other vertex, we use the **AllPairsDistance** command. This is an implementation of the Floyd-Warshall algorithm (also known as simply the Floyd algorithm), which is described in Algorithm 2 in the Exercises of Section 10.6.

```
> AllPairsDistance(Ex2);
```

$$\begin{bmatrix} 0 & 2 & 3 & 5 & 4 & 7 \\ 2 & 0 & 5 & 3 & 2 & 5 \\ 3 & 5 & 0 & 6 & 5 & 8 \\ 5 & 3 & 6 & 0 & 1 & 2 \\ 4 & 2 & 5 & 1 & 0 & 3 \\ 7 & 5 & 8 & 2 & 3 & 0 \end{bmatrix} \qquad (10.114)$$

Note that the command returned a matrix. The (i, j) entry in this matrix is the shortest distance from vertex i to vertex j.

Finally, Maple provides a **`TravelingSalesman`** command for solving the traveling salesperson problem on a given graph. Given a graph, the procedure returns two objects: a number representing the minimum possible length of a Hamilton circuit and the list of vertices representing the minimal circuit.

```
> TravelingSalesman(Ex2);
                    21, ["a", "b", "d", "z", "e", "c", "a"]                (10.115)
```

10.7 Planar Graphs

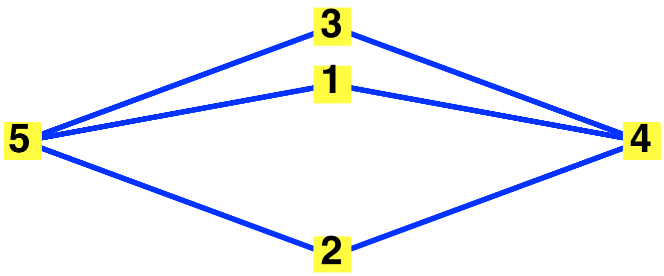
This section explains how Maple can be used to explore the question of whether a graph is planar. We begin with a brief description of Maple's built-in functions. We then discuss how to use Maple to manipulate graphs in order to produce homeomorphic graphs and to apply Kuratowski's Theorem.

Maple includes the command **`IsPlanar`**, which returns true if and only if the given graph is a planar graph. For example, we can check that the graph $K_{3,2}$ is planar, but that $K_{3,3}$ is not.

```
> IsPlanar(CompleteGraph(3,2));
                               true                                        (10.116)
> IsPlanar(CompleteGraph(3,3));
                               false                                       (10.117)
```

Also, as mentioned above, for those graphs that are planar, the **`DrawGraph`** command includes the option to draw them as such, using the planar style.

```
> DrawGraph(CompleteGraph(3,2),style=planar);
```



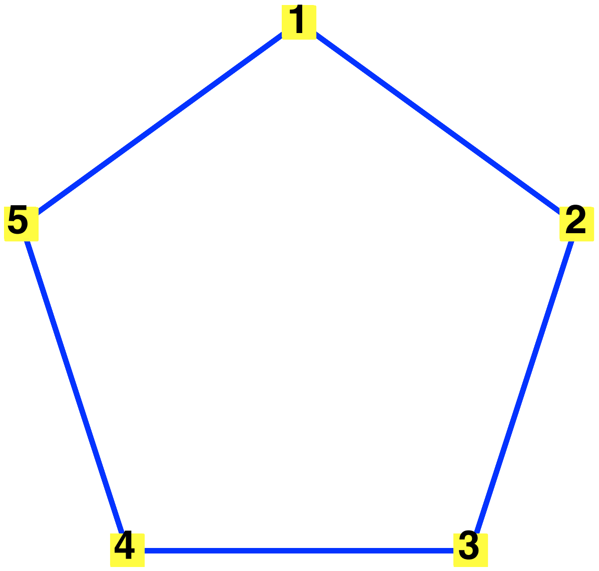
For graphs that are not planar, the planar style will cause an error to be raised.

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. Maple includes a command, **Subdivide**, for achieving this effect. If we apply this command to a graph and one of its edges, it returns a new graph obtained by performing an elementary subdivision on the given edge.

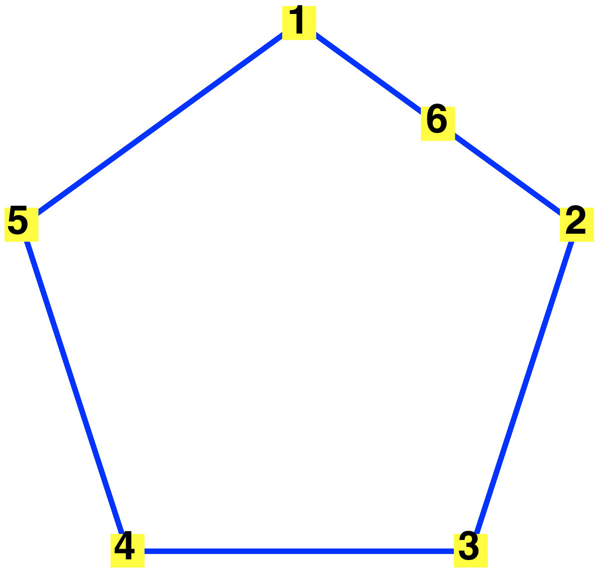
```
> SubdivideEx := CycleGraph(5);
SubdivideEx := Graph 73: an undirected unweighted graph with 5 vertices and 5 edge(s) (10.118)
```

```
> DrawGraph(SubdivideEx);
```



```
> SubdivideEx2 := Subdivide(SubdivideEx,{1,2});
SubdivideEx2 :=                                                         (10.119)
     Graph 74: an undirected unweighted graph with 6 vertices and 6 edge(s)
```

```
> DrawGraph(SubdivideEx2);
```

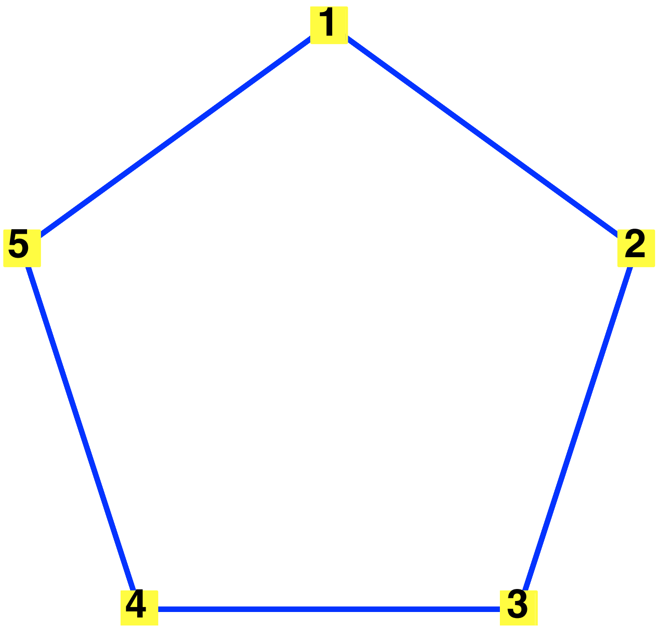


The inverse operation of elementary subdivision is referred to as smoothing. To be precise, let v be a vertex of degree 2 with neighbors u and w and such that u and w are not adjacent. We smooth the vertex v by deleting v and the edges incident to it and adding the edge $\{u, w\}$. Below we have created a procedure to implement smoothing. (Note that Maple's `Contract` command is more general than smoothing. The benefits of creating the `Smooth` procedure are that it is explicitly the inverse of elementary subdivision and that it is more natural, in this context, to think about smoothing the vertex rather than contracting one of the incident edges.)

```
> Smooth := proc(G::Graph,v)
    local e, H;
    e := {op(Neighbors(G,v))};
    if (Degree(G,v) <> 2) or (e in Edges(G)) then
      return FAIL;
    else
      H := DeleteVertex(G,v);
      AddEdge(H,e);
    end if;
    return H;
  end proc:
```

```
> SubdivideEx3 := Smooth(SubdivideEx2,6);
SubdivideEx3 :=                                                    (10.120)
    Graph 75: an undirected unweighted graph with 5 vertices and 5 edge(s)
```

```
> DrawGraph(SubdivideEx3);
```



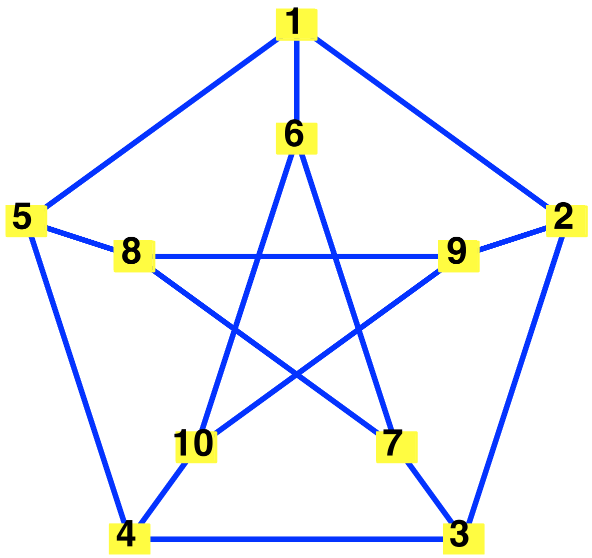
The textbook defines graphs to be homeomorphic if they can be obtained from the same graph from a sequence of elementary subdivisions. It is clear that if $G_1, G_2, G_3, \ldots, G_n$ is a sequence of graphs, each of which can be obtained from the previous by an elementary subdivision, then $G_n, \ldots, G_3, G_2, G_1$ 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 commands above and those for creating subgraphs, we can use Maple to manipulate a graph and confirm that it is nonplanar using Kuratowski's Theorem. We will illustrate this with the Petersen graph.

```
> petersen := SpecialGraphs[PetersenGraph]();
 petersen := Graph 76: an undirected unweighted graph with 10 vertices and 15 edge(s)  (10.121)
```

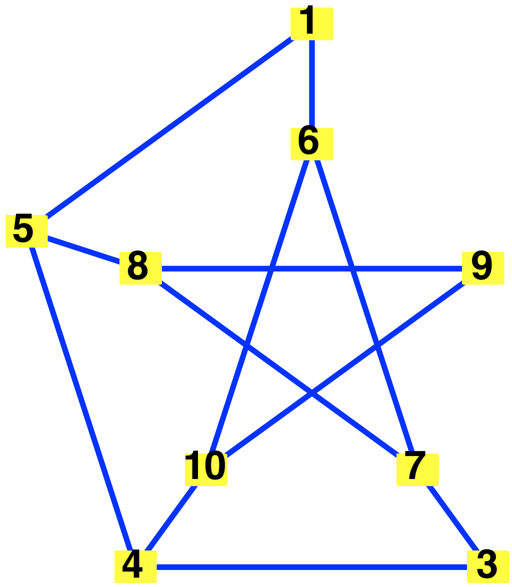
```
> DrawGraph(petersen);
```



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

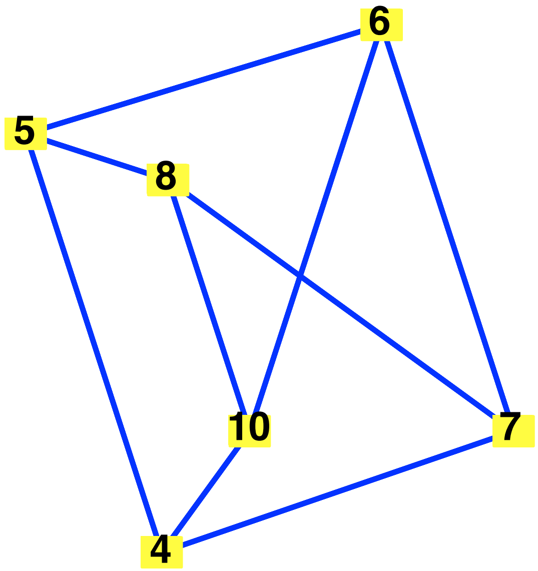
```
> petersen1 := DeleteVertex(petersen,2);
 petersen1 := Graph 77: an undirected unweighted graph with 9 vertices and 12 edge(s)  (10.122)
```

```
> DrawGraph(petersen1);
```



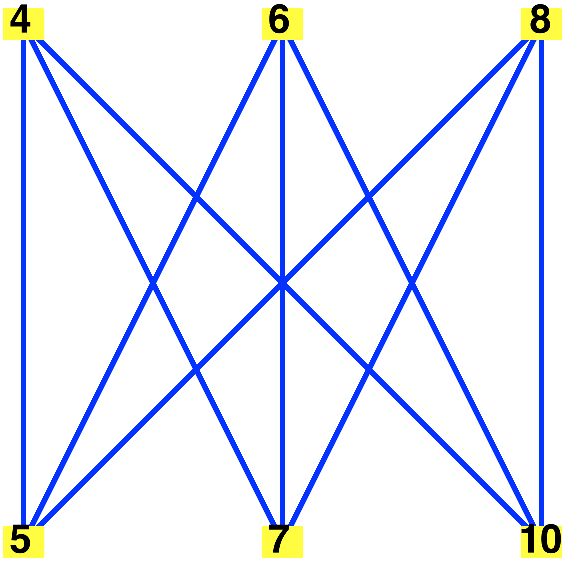
Now we notice that there are three vertices that are smoothable: 1, 3, and 9. That is to say, those three vertices have degree 2 and their neighbors are not adjacent.

```
> petersen2 := Smooth(petersen1,1):
> petersen3 := Smooth(petersen2,3):
> petersen4 := Smooth(petersen3,9):
> DrawGraph(petersen4);
```



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 check that it is bipartite and then have Maple draw it in that style.

```
> IsBipartite(petersen4);
                                  true                                  (10.123)
> DrawGraph(petersen4,style=bipartite);
```



It is clear from inspection that this is $K_{3,3}$ and so we have demonstrated 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.

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

Maple's Command

Maple provides a `ChromaticNumber` command that uses a sophisticated backtracking technique for computing the chromatic number of a graph.

Given a graph, the `ChromaticNumber` command will report the minimal number of colors needed to color that graph.

```
> CNExample := SpecialGraphs[WheelGraph](5);
CNExample :=                                                        (10.124)
    Graph 78: an undirected unweighted graph with 6 vertices and 10 edge(s)
> ChromaticNumber(CNExample);
                                   4                                (10.125)
```

If you provide a variable name as a second argument to the command, Maple will store a list of lists of vertices. These lists indicate which vertices should be assigned the same color.

```
> ChromaticNumber(CNExample,'CNClasses');
                                   4                                (10.126)
> CNClasses;
                      [[0], [1, 3], [2, 4], [5]]                    (10.127)
```

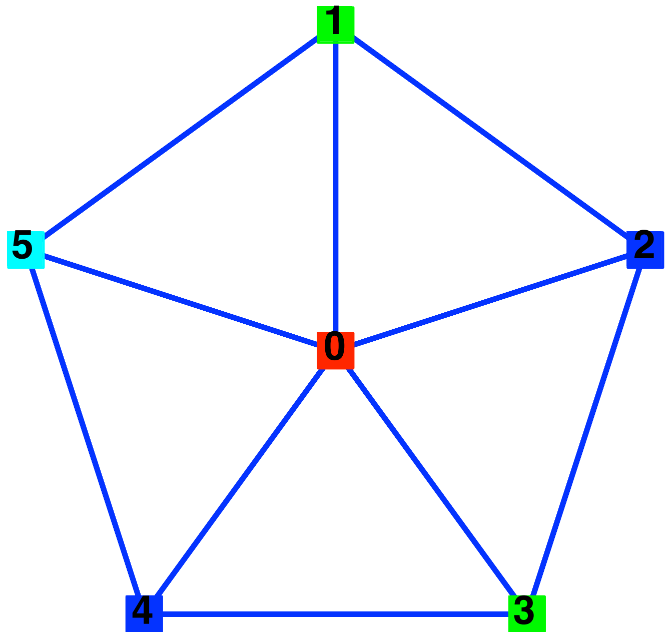
This output indicates that vertex 0 should be given one color, vertex 1 and 3 should be assigned a second color, vertex 2 and 4 a third color, and vertex 5 should be painted with the final color.

We can write a short procedure to display the graph with the vertices appropriately colored. Our procedure will call the `HighlightVertex` command with a list of vertices and a single color. This form of the command causes all of the vertices in the list to be shaded with the specified color.

```
> CNColor := proc(G::Graph, CNout::list, colors::list)
    local i;
    if nops(CNout) <> nops(colors) then
      print("You must give one color for each vertex class.");
      return FAIL;
    end if;
    for i from 1 to nops(CNout) do
      GraphTheory[HighlightVertex](G,CNout[i],colors[i]);
    end do;
    GraphTheory[DrawGraph](G);
  end proc:
```

This procedure requires three arguments: the graph, the output from `ChromaticNumber`, and a list of color names.

```
> CNColor(CNExample,CNClasses,[red,green,blue,cyan]);
```



A Greedy Coloring Algorithm

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

For our graph coloring procedure, this is further complicated by the fact that the procedure that we pass to the `sort` command must depend on the graph. We will create a functional operator that returns a boolean-valued procedure associated to the given graph.

```
> MakeSorter := G -> proc(v,w)
    if GraphTheory[Degree](G,v) > GraphTheory[Degree](G,w) then
      return true;
    else
      return false;
    end if;
```

```
  end proc:
```

Applying `MakeSorter` to a graph returns a procedure that can be used as the optional argument to `sort`.

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.

```
> ColorList := [red, green, blue, magenta, orange, turquoise,
  violet, cyan, brown, black];
 ColorList := [red, green, blue, magenta, orange, turquoise, violet, cyan, brown, black]  (10.128)
```

Now we will implement the greedy coloring algorithm.

```
> GColor := proc(G::Graph)
    local Sorter, V, currentColor, excludeSet, i;
    Sorter := MakeSorter(G);
    V := sort(Vertices(G),Sorter);
    for currentColor from 1 to nops(ColorList) do
      HighlightVertex(G,V[1],ColorList[currentColor]);
      excludeSet := {op(Neighbors(G,V[1]))};
      V := subsop(1=NULL,V);
      i := 1;
      while i <= nops(V) do
        if not (V[i] in excludeSet) then
          HighlightVertex(G,V[i],ColorList[currentColor]);
          excludeSet:=excludeSet union {op(Neighbors(G,V[i]))};
          V := subsop(i=NULL,V);
        else
          i := i + 1;
        end if;
      end do;
      if V = [] then
        break;
      end if;
    end do;
    if V <> [] then
      print("Insufficiently many colors");
      return FAIL;
      HighlightVertex(G,Vertices(G),yellow);
    else
      DrawGraph(G);
    end if;
  end proc:
```

Note that the set `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 `subsop(i=NULL,V)`. The `subsop` command is used to substitute a value in a list at a specified index. In this case, we're substituting the value `NULL` in the list at index `i`, which has the effect of removing it from the list.

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

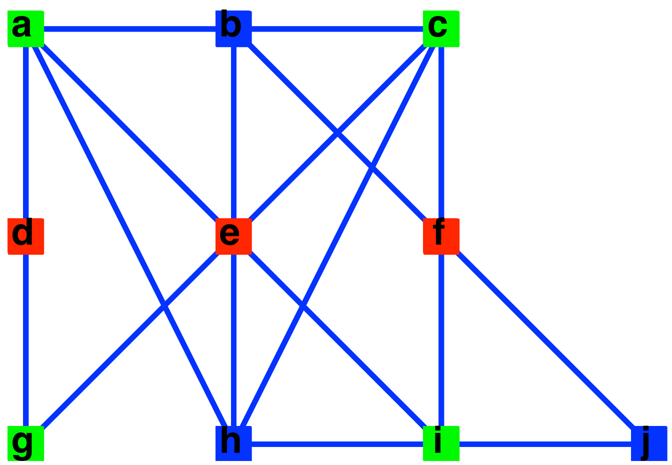
The index `i`, which controls the while loop, is incremented in the else clause of the if statement that tests to see if a vertex can be assigned the color. If the vertex at index `i` is assigned the color, then it is removed from the list `V`, and thus the index `i` refers to a different vertex (the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

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

```
> Exercise29 := Graph({{"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"}});
```

```
Exercise29 :=                                                    (10.129)
    Graph 79: an undirected unweighted graph with 10 vertices and 18 edge(s)
```

```
> SetVertexPositions(Exercise29,[[0,2],[1,2],[2,2],[0,1],[1,1],
  [2,1],[0,0],[1,0],[2,0],[3,0]]);
> GColor(Exercise29);
```



Solutions to Computer Projects and Computations and Explorations

Computations and Explorations 1

Display all simple graphs with four vertices.

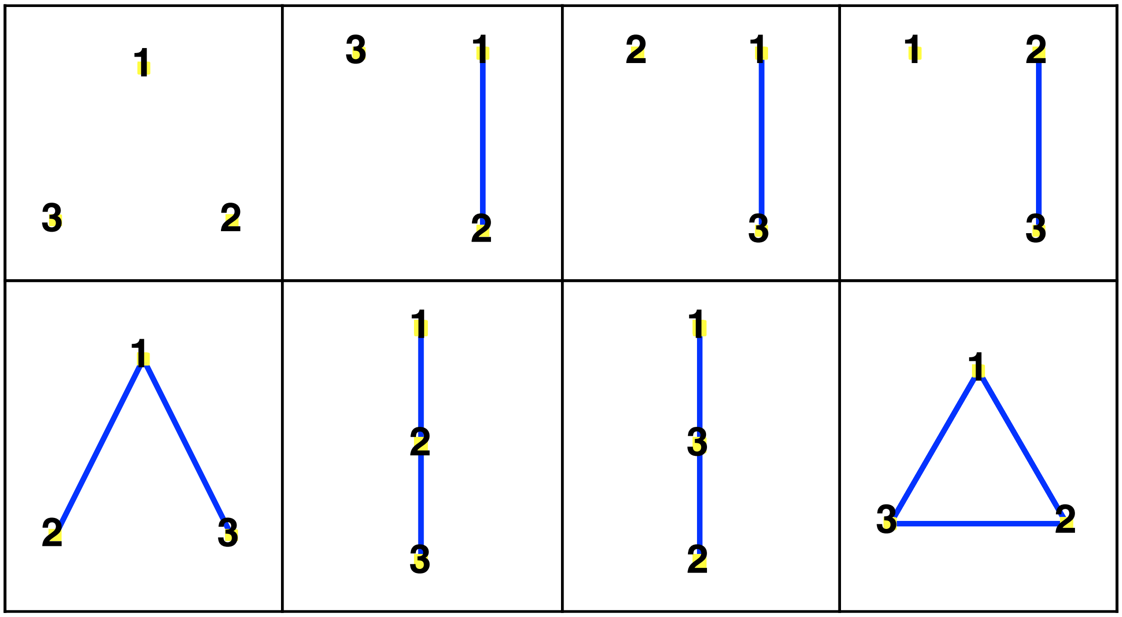
Solution: To solve this problem, we will generate all possible edge sets and then construct the graphs based on these edge sets. The possible edge sets are all of the subsets of the set of all possible edges, which we obtain from the complete graph on the vertices. We will generalize the question and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

```
> AllGraphs := proc(n::posint)
    local A, V, E, powerE, i, G;
    uses GraphTheory;
    A := {};
    V := [seq(i,i=1..n)];
    E := Edges(CompleteGraph(n));
    powerE := combinat[powerset](E);
```

```
      for i from 1 to nops(powerE) do
        G[i] := Graph(V,powerE[i]);
        A := A union {G[i]};
      end do;
      return A;
    end proc:
```

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.

```
> AllGraphs3 := AllGraphs(3):
> DrawGraph(AllGraphs3,width=4);
```



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, each time a graph is generated, we compare it to the graphs that have already been included using `IsIsomorphic`.

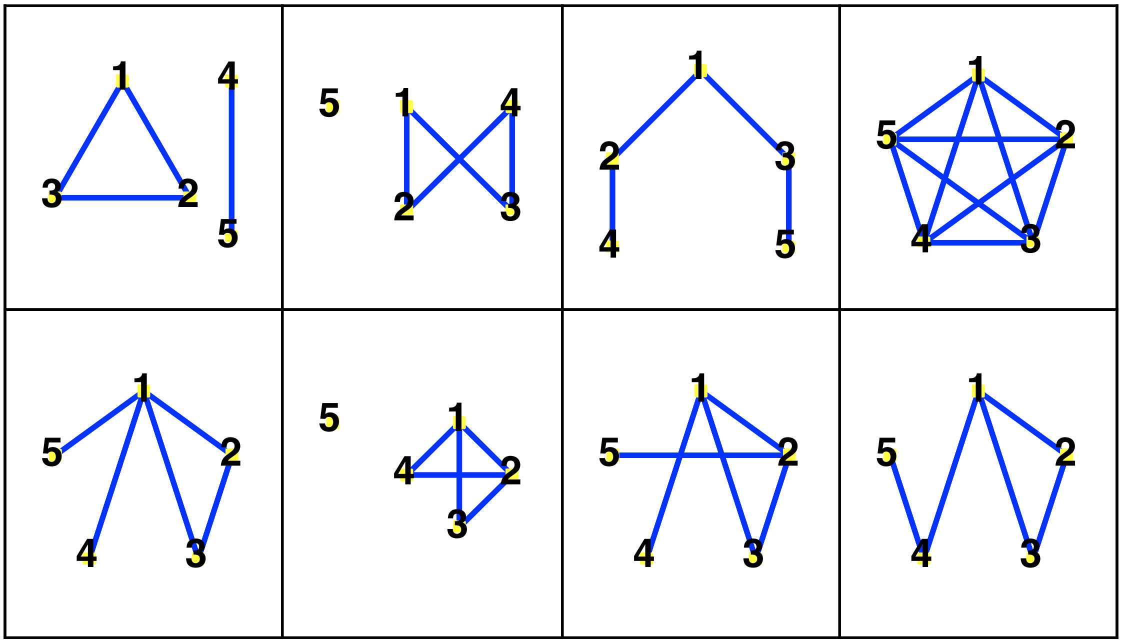
```
> NonIsoGraphs := proc(n::posint)
    local A, V, E, powerE, i, G, j, notisomorphic;
    uses GraphTheory;
    A := {};
    V := [seq(i,i=1..n)];
    E := Edges(CompleteGraph(n));
    powerE := combinat[powerset](E);
    for i from 1 to nops(powerE) do
      G[i] := Graph(V,powerE[i]);
      notisomorphic := true;
      for j from 1 to nops(A) do
        if IsIsomorphic(A[j],G[i]) then
          notisomorphic := false;
          break;
        end if;
        end do;
        if notisomorphic then
          A := A union {G[i]};
        end if;
      end do;
      return A;
    end proc:
```

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

```
> NonIso5 := NonIsoGraphs(5):
> nops(NonIso5);
                                 34                              (10.130)
```

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

```
> DrawGraph([seq(NonIso5[i],i=1..8)],width=4);
```

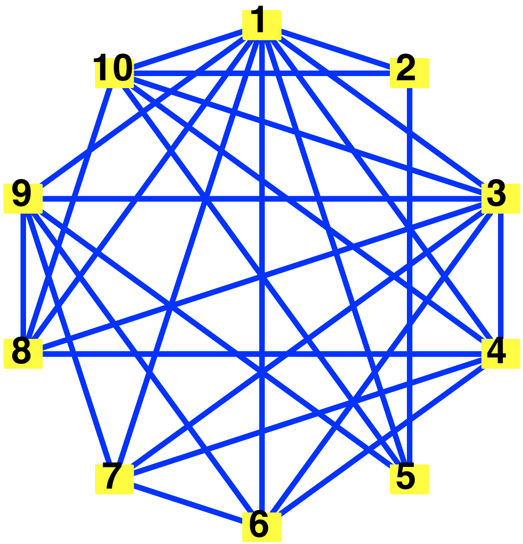


Computations and Explorations 9

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** command in the **RandomGraphs** subpackage. By passing this command a number of vertices and a probability between 0 and 1, it produces a graph with the given number of vertices and with each possible edge present with the given probability. To display a random graph on 10 vertices with each edge as likely to appear as not, we use the following command.

```
> DrawGraph(RandomGraphs[RandomGraph](10,.5));
```

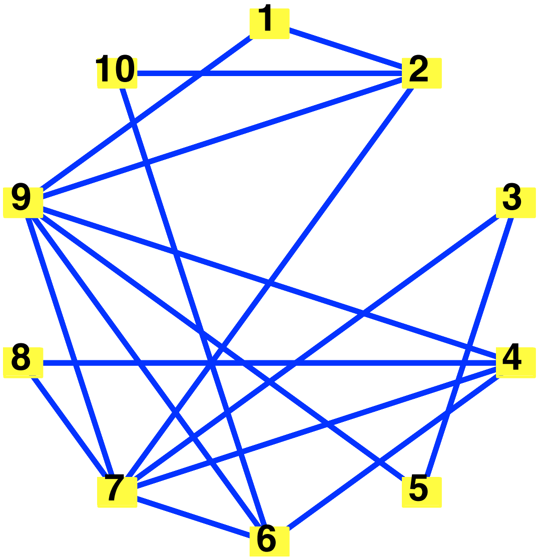


Recall the description of the **IsEulerian** command. When this command is given two arguments, specifically, a graph and an optional variable name, if there is an Euler circuit, then the command returns true and stores the circuit in the variable.

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 **IsEulerian**. As long as the randomly generated graph does not have an Euler circuit, we continue generating new random graphs. We display the path using the `animatePath` procedure we created in Section 10.5.

```
> GenEuler := proc(n::posint)
    local G, trail;
    G := GraphTheory[RandomGraphs][RandomGraph](n,.5);
    while not GraphTheory[IsEulerian](G,'trail') do
      G := GraphTheory[RandomGraphs][RandomGraph](n,.5);
    end do;
    animatePath(G,[op(trail)]);
  end proc:
```

```
> GenEuler(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 procedure that generates a number of random graphs of the specified size and counts the number that are connected. We use the `RandomGraph` command to create the random graphs and the `IsConnected` command to test them for connectivity.

```
> ConnectedProbability := proc(verts::posint, total::posint)
    local G, i, count;
    count := 0;
    for i from 1 to total do
      G := GraphTheory[RandomGraphs][RandomGraph](verts,.5);
      if GraphTheory[IsConnected](G) then
        count := count + 1;
      end if;
    end do;
    return count/total;
  end proc:
> [seq(ConnectedProbability(i,100),i=1..10)];
```

$$\left[1, \frac{53}{100}, \frac{2}{5}, \frac{13}{25}, \frac{13}{20}, \frac{39}{50}, \frac{23}{25}, \frac{19}{20}, \frac{47}{50}, \frac{49}{50}\right] \tag{10.131}$$

Exercises

Exercise 1. Write a Maple procedure to find *all* maximal matchings for a bipartite graph.

Exercise 2. Write Maple procedures for calculating the adjacency and incidence matrices for a pseudograph.

Exercise 3. Write a Maple procedure for creating a pseudograph from an incidence matrix.

Exercise 4. Write a Maple procedure to find all of the minimal edge cuts of a given graph.

Exercise 5. Write a Maple procedure to count the number of Hamilton circuits in a simple graph.

Exercise 6. Write a Maple procedure 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.

Exercise 7. Use Maple to construct all regular graphs of degree *n*, given a positive integer *n*. (Regular is defined in the Exercises for Section 10.2.)

Exercise 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 Maple procedure that calculates the local vertex connectivity of a graph and a pair of its vertices.

Exercise 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 Maple procedure that calculates the local edge connectivity of a graph and a pair of its vertices.

Exercise 10. Write a Maple procedure that computes the thickness of a nonplanar simple graph (see the Exercises in Section 10.7 for a definition of thickness).

Exercise 11. Write a Maple procedure for finding an orientation of a simple graph. (An orientation of a graph is defined in the Supplementary Exercises of Chapter 10.)

Exercise 12. Write a Maple procedure for finding the bandwidth of a simple graph. (The bandwidth of a graph is defined in the Supplementary Exercises of Chapter 10.)

Exercise 13. Write a Maple procedure 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.)

Exercise 14. Use Maple 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.

Exercise 15. Write a Maple procedure 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 procedure to display the self-complementary graphs for as large a n as possible.

Exercise 16. Write a Maple procedure 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.

Exercise 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 Maple procedure for determining whether a sequence of positive integers is graphic and, if it is, to construct a graph with this degree sequence.