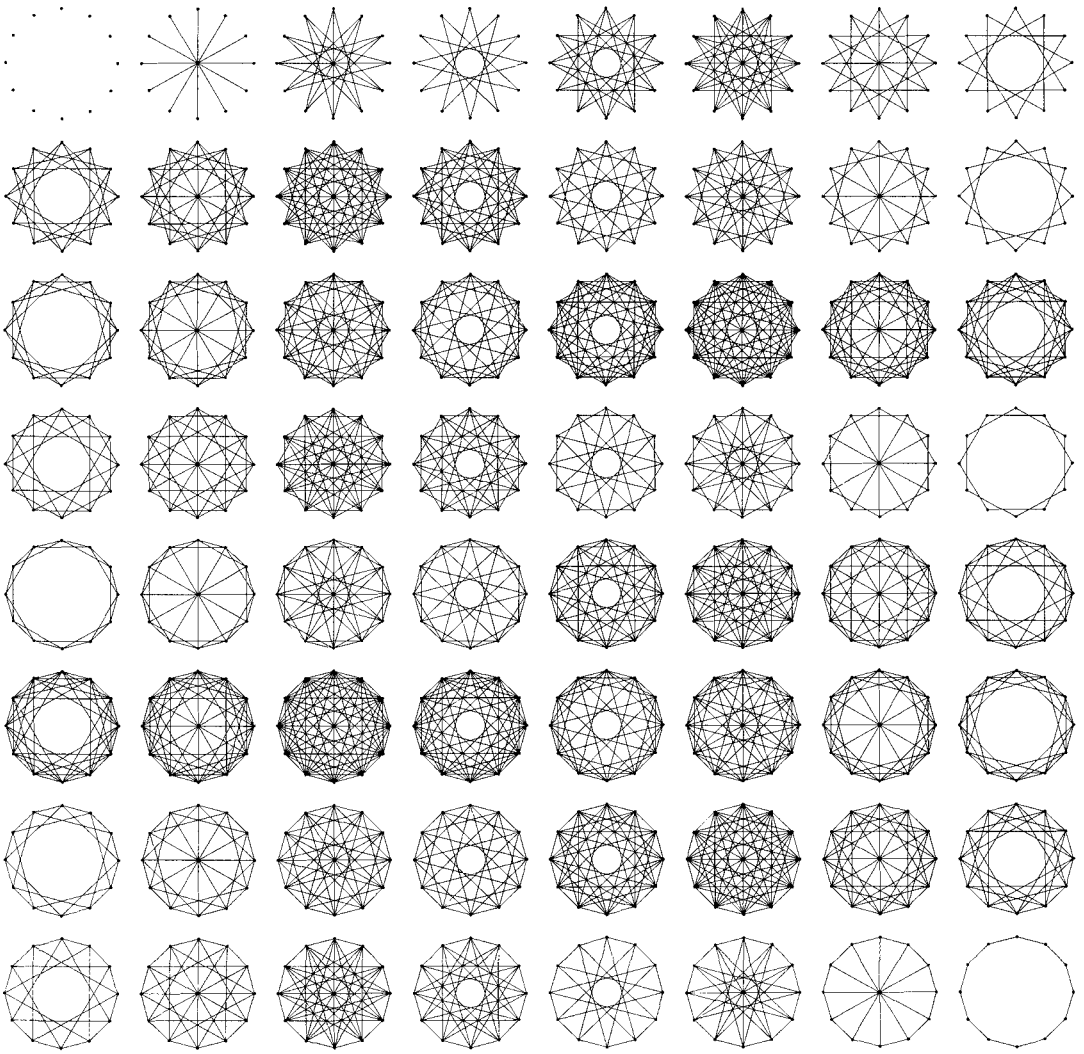
6. Generating Graphs



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

This chapter is devoted to constructing graphs. Using the functions provided in this chapter, a user can construct a large variety of graphs. Circulant graphs, grid graphs, hypercubes, and butterfly graphs are just a few of the graphs you will encounter. Some of the graph classes we provide are random, and these are especially suited for testing graph algorithms. We also provide a large number of graph operations that build new graphs from old ones. Union, join, product, and contract are some of the operations you will encounter. In addition, we provide an easy way of constructing graphs from binary relations and functions. Many real-life graphs express some underlying binary relation, and being able to easily construct such graphs is one of the highlights of *Combinatorica*.

The properties of the graphs constructed in this chapter will motivate the graph invariants that we compute in later chapters. Many of these graphs are truly beautiful when drawn properly, and they provide a wide range of structures to manipulate and study.

About the illustration overleaf:

Circulant graphs are highly symmetrical and parametrized by subsets of vertices. Here are all the 2^6 circulant graphs on 12 vertices, built from the list of all 6-element subsets with the command:

```
ShowGraphArray[ Partition[Map[(CirculantGraph[12,#])&, Subsets[6]],8]]
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

6.1 Building Graphs from Other Graphs

This section presents operations that build graphs from other graphs. Many of the parametrized graphs in this chapter can be constructed using these operations. Other examples appear in [CM78].

■ 6.1.1 Contracting Vertices

Contracting a pair of vertices, v_1 and v_2, replaces them by one vertex v such that v is adjacent to anything v_1 or v_2 had been. It does not matter whether v_1 and v_2 are connected by an edge; if they are, then the edge disappears when v_1 and v_2 are contracted. The function `Contract` generalizes this and can shrink two or more vertices in a graph into one.

`Contract` runs in linear time by constructing a mapping from the vertices of the original graph to the vertices of the contracted graph. Suppose that we are given an n-vertex graph $G = (V, E)$ and a subset L with k vertices to contract. Contracting G gives a graph H with $n-k+1$ vertices. Each vertex v in L is mapped to vertex $n-k+1$ in H. Every other vertex v in G is mapped to vertex $v-i$ in H if there are i vertices in L smaller than v. This mapping is constructed in $\Theta(n)$ time. We then use this mapping to create the edges of H, in time proportional to the number of edges in G.

```
Contract[g_Graph, l_List] :=
     Module[{v = Vertices[g, All], t = Table[0, {V[g]}],
            cnt = 0, last = V[g] - Length[l] + 1},
           Do[If[MemberQ[l, k], cnt++; t[[k]] = last, t[[k]] = k - cnt], {k, V[g]}];
           Graph[
             DeleteCases[Edges[g, All] /. {{x_Integer, y_Integer}, opts___?OptionQ}
                                 :> {Sort[{t[[x]], t[[y]]}], opts}, {{last, last}, opts___?OptionQ}
             ],
             Append[v[[Complement[Range[Length[v]], l]]],
                  {Apply[Plus, Map[First, v[[l]]]]/Length[l]}
             ],
             Apply[Sequence, GraphOptions[g]]
           ]
     ]
```

Contracting Vertices in a Graph

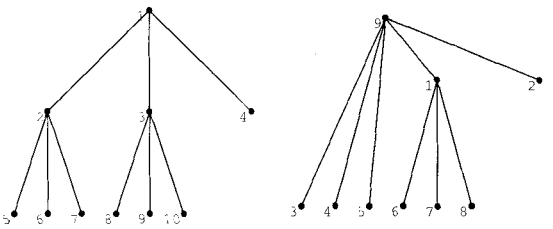
This loads the package.

```
In[1]:= <<DiscreteMath`Combinatorica`
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

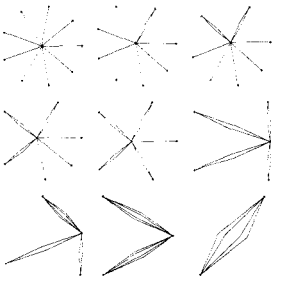
Contracting edge (1,2) shrinks the number of vertices and edges by one, because the contracted edge disappears. Note the renumbering of vertices: The new vertex is numbered 9 and all other vertices slide up by 2 in the ordering. `Contract` maintains the embedding of the old graph: The new vertex is placed at the midpoint of the two contracted vertices, with the locations of all other vertices unchanged.

```
In[2]:= ShowGraphArray[{g = CompleteKaryTree[10, 3], Contract[g, {1, 2}]},
          VertexNumber -> True];
```



Contracting vertices will create multiple edges whenever contracted vertices have a common neighbor. Thus contract operations can decrease the number of vertices in the graph without decreasing the number of edges.

```
In[3]:= g = Star[10];
        ShowGraphArray[Partition[NestList[Contract[#, {1, 2}] &, g, 9], 3]];
```



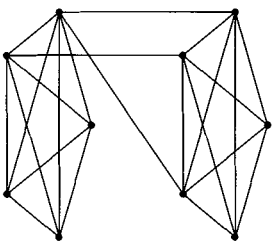
Graph contraction is useful in counting the number of spanning trees of a graph. A *spanning tree* of graph $G = (V, E)$ is a tree whose vertex set equals V and whose edges are all contained in E. The number of spanning trees of G, denoted $\tau(G)$, satisfies the recurrence relation $\tau(G) = \tau(G - e) + \tau(G \cdot e)$. Here $G - e$ and $G \cdot e$ respectively denote the graphs obtained from G by deleting edge e and by contracting the endpoints of edge e.

```
In[5]:= g = RandomGraph[20, .5]; e = First[Edges[g]];
        {NumberOfSpanningTrees[g],
        NumberOfSpanningTrees[DeleteEdges[g, {e}]] +
        NumberOfSpanningTrees[Contract[g, e]]}
Out[6]= {14021885085330334, 14021885085330334}
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

A *minimum cut* of a graph is a smallest set of edges whose removal disconnects the graph. The set of three edges that connect the K_5 on the left to the K_5 on the right is a minimum cut of this graph. Computing a minimum cut of a graph is an important optimization problem.

```
In[7]:= ShowGraph[h1= h2=AddEdges[GraphUnion[g=CompleteGraph[5], g],
        {{1,6}, {1,8}, {2,7}}]];
```

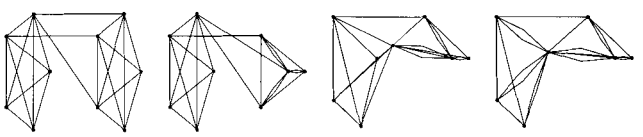


The size of a minimum cut set is the graph's *edge connectivity*.

```
In[8]:= EdgeConnectivity[h1]
Out[8]= 3
```

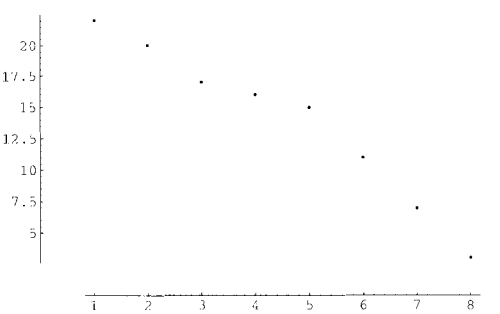
Repeatedly contracting randomly chosen edges yields a beautiful randomized algorithm for computing a minimum cut [KS96]. Observe that contracting an edge does not decrease the size of a minimum cut, since any cut in G' is also a cut in the original graph G. Here, contracting an edge in one of the K_5's maintains the size of a minimum cut, and contracting an edge between the K_5's increases the minimum cut.

```
In[9]:= ShowGraphArray[Join[{h1}, Table[el=Edges[h1]; h1 = Contract[h1,
        el[[Random[Integer, {1,Length[el]}]]]], {3}]]];
```



Every edge contraction decreases the number of vertices by one, so after $n-2$ edge contractions we get a 2-vertex graph. The number of edges in this multigraph is an upper bound on the size of a minimum cut in the original graph. This upper bound is not always tight, but the smallest graph encountered after repeating this algorithm polynomially many times is very likely to give a minimum cut.

```
In[10]:= ListPlot[Table[el = Edges[h2]; h2 = Contract[h2,
         el[[Random[Integer, {1, Length[el]}]]]]; M[h2], {8}],
         AxesOrigin->{0,0}];
```



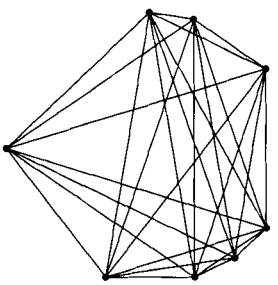
https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

■ 6.1.2 Inducing and Permuting Subgraphs

An *induced subgraph* of a graph G is a subset of the vertices of G together with any edges whose endpoints are both in this subset. Deleting a vertex from a graph is identical to inducing a subgraph of the remaining $n-1$ vertices. *Combinatorica* provides a function `InduceSubgraph` that takes a graph G and a subset S of the vertices of G and returns the subgraph of G induced by S. `InduceSubgraph` calls a more general function `PermuteSubgraph`, which not only induces a subgraph, but permutes the embedding of the graph according to the given permutation.

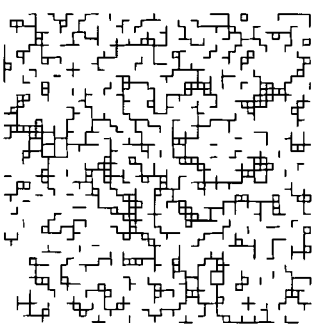
Any subset of the vertices in a complete graph defines a clique. This drawing presents an interesting illusion, for although the points seem irregularly spaced, they all are defined as lying on a circle.

```
In[11]:= ShowGraph[InduceSubgraph[CompleteGraph[20], RandomSubset[20]]];
```



A random induced subgraph of a grid graph looks like a maze. A *connected component* of a graph is a maximal connected subgraph. Random graph theory concerns quantities like the expected number of connected components and the expected size of the largest component in a random graph.

```
In[12]:= ShowGraph[g = InduceSubgraph[GridGraph[50, 50],
         RandomSubset[2500]], VertexStyle -> Disk[0]];
```



What is the expected number of connected components in a random induced subgraph of an $n \times n$ grid graph?

```
In[13]:= Length[ ConnectedComponents[g] ]
Out[13]= 173
```

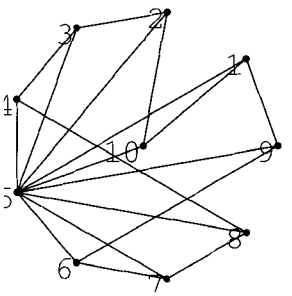
https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

This experiment hints at what the average number of connected components in a 20×20 grid graph might be. `ConnectedComponents` is a function to compute the connected components of a graph (see Section 7.2.1).

```
In[14]:= Table[Length[ConnectedComponents[InduceSubgraph[GridGraph[20, 20],
           RandomSubset[400]]]], {5}]
Out[14]= {37, 39, 31, 35, 43}
```

This gives an unfamiliar embedding of the 10-vertex wheel graph. The same locations are used for the ten vertices, but the vertices themselves are moved around according to the provided *permutation*. `PermuteSubgraph` can be used to generate new embeddings of graphs.

```
In[15]:= ShowGraph[h = PermuteSubgraph[Wheel[10], RandomPermutation[10]],
           VertexNumber -> True, TextStyle -> {FontSize -> 12}];
```

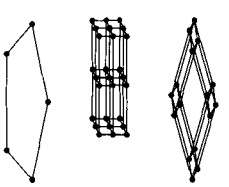


■ 6.1.3 Unions and Intersections

Graphs are sets of vertices and edges, and the most important operations on sets are union and intersection. The *union* operation takes two or more graphs and returns a graph that is formed by taking the union of the vertices and edges of the graphs. `GraphUnion` first normalizes the embeddings of the given graphs, to ensure that their drawings are roughly the same size. It then puts together the edges and vertices of the given graphs, making sure that each successive graph appears to the right of the previous one.

Here is the union of a 5-cycle, $3 \times 3 \times 3$ grid graph and a four-dimensional hypercube. The component graphs of the union are placed in order from left to right. Each component graph is normalized to ensure that the relative sizes of the components are similar.

```
In[16]:= ShowGraph[g=GraphUnion[Cycle[5], GridGraph[3,3,3], Hypercube[4]]];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The union of graphs is always disconnected.

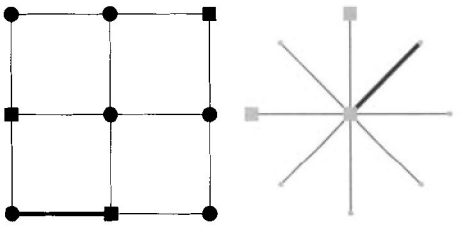
```
In[17]:= ConnectedComponents[g]
Out[17]= {{1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
     20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32},
    {33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
     46, 47, 48}}
```

This graph contains 48 vertices, 91 edges, and 3 connected components.

```
In[18]:= {V[g], M[g], Length[ConnectedComponents[g]]}
Out[18]= {48, 91, 3}
```

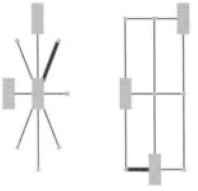
Here are two graphs with different local and global options set. We will use these graphs as examples to show which graph attributes are preserved by graph operations.

```
In[19]:= g = SetGraphOptions[GridGraph[3, 3], {{2,4,9,
          VertexStyle->Box[Large]}, {{1,2}, EdgeStyle->Thick}},
          VertexStyle->Disk[Large]];
          h = SetGraphOptions[Star[9], {{2,4,9, VertexStyle->Box[Large]},
          {{1,9}, EdgeStyle->Thick}}, VertexColor->Gray];
          ShowGraphArray[{g, h}];
```



`GraphUnion` inherits global options from the first graph while preserving all local options. Here all vertices in the union appear `Gray` because of the global vertex option associated with the first graph. Local options that alter the thickness of edges and size of vertices are preserved for both graphs.

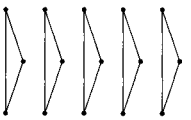
```
In[22]:= ShowGraph[GraphUnion[h, g]];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The other form of GraphUnion specifies the number of copies of the graph to union.

```
In[23]:= ShowGraph[GraphUnion[5, CompleteGraph[3]]];
```



Why does this function call return unevaluated? The first two graphs are directed while the third is undirected. GraphUnion requires all its argument graphs to be uniformly directed or undirected. Mixed input is returned unevaluated.

```
In[24]:= GraphUnion[CompleteGraph[6, Type -> Directed],
           CompleteGraph[5, Type -> Directed], CompleteGraph[5]]
Out[24]= GraphUnion[-Graph:<30, 6, Directed>-,
    -Graph:<20, 5, Directed>-, -Graph:<10, 5, Undirected>-]
```

The *intersection* of two graphs with the same number of vertices is constructed by taking the intersection of the edges of the graphs.

```
GraphIntersection[g_Graph] := g

GraphIntersection[g_Graph,h_Graph] :=
      Module[{e = Intersection[Edges[g], Edges[h]]},
            ChangeEdges[g, Select[Edges[g, All], (MemberQ[e, First[#]]) &]]
      ] /; (V[g] == V[h]) && (UndirectedQ[g] == UndirectedQ[h])

GraphIntersection[g_Graph,h_Graph, l__Graph] := GraphIntersection[GraphIntersection[g, h], l]
```

The Intersection of Graphs

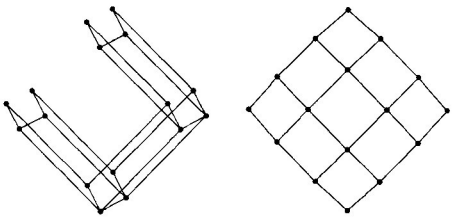
Intersection with K_n is an identity operation for any graph of order n.

```
In[25]:= IdenticalQ[ GraphIntersection[Wheel[10], CompleteGraph[10]],
           Wheel[10]]
Out[25]= True
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

What is the intersection of a hypercube and grid graph of equal sizes? The SpringEmbedding of the result reveals the original grid graph! Thus the grid graph is a subgraph of the hypercube.

```
In[26]:= ShowGraphArray[{gh=GraphIntersection[ Hypercube[4],
           GridGraph[4,4]], SpringEmbedding[gh,200] }];
```



■ 6.1.4 Sums and Differences

Since graphs can be represented by adjacency matrices, they can be added, subtracted, and multiplied in a meaningful way, provided they have the same number of vertices.

The sum of a graph and its complement gives the complete graph.

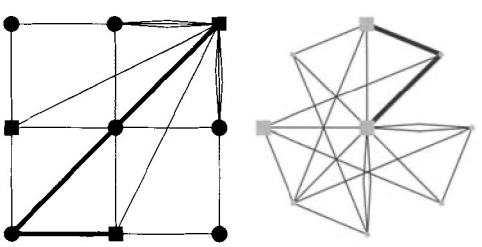
```
In[27]:= CompleteQ[GraphSum[Cycle[10], GraphComplement[Cycle[10]]]]
Out[27]= True
```

The difference of a graph and itself gives the empty graph.

```
In[28]:= EmptyQ[GraphDifference[Cycle[10],Cycle[10]] ]
Out[28]= True
```

GraphSum's rule of inheritance takes the edges of the second graph and adds them to the first graph. Thus all global options come from the first input graph; each edge in the graph sum belongs to one of the input graphs and preserves its local options; each vertex inherits its local options from the first input graph.

```
In[29]:= ShowGraphArray[{GraphSum[g,h], GraphSum[h,g]}];
```



■ 6.1.5 Joins of Graphs

The *join* of two graphs is their union, with the addition of all edges spanning the different graphs. Many of the graphs we have seen as examples and will implement in this chapter can be specified as the join of two graphs, such as complete bipartite graphs, stars, and wheels.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

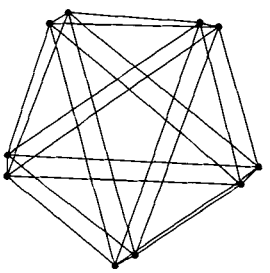
The *Cartesian product* $A \times B$ is the set of element pairs such that one element is from A and one is from B. The edges that get added to the union of the two graphs are exactly the Cartesian product of the two vertex sets.

```
GraphJoin[g_Graph] := g

GraphJoin[g_Graph,h_Graph] :=
       AddEdges[GraphUnion[g, h],
              CartesianProduct[Range[V[g]],Range[V[h]]+V[g]]
       ] /; (UndirectedQ[g] == UndirectedQ[h])

GraphJoin[g_Graph,h_Graph, l__Graph] := GraphJoin[GraphJoin[g, h], l]

CartesianProduct[a_List, b_List] := Flatten[Outer[List, a, b, 1, 1], 1]
```

The Join of Graphs

Complete k-partite graphs such as $K_{5,5}$ are naturally described in terms of `GraphJoin`. Here we show a "spring embedding" of the graph because the default embedding of the join looks somewhat ugly.

```
In[30]:= ShowGraph[SpringEmbedding[
           GraphJoin[EmptyGraph[5], EmptyGraph[5]]]];
```



A wheel is the join of a cycle and a single vertex. A star is the join of an empty graph and a single vertex.

```
In[31]:= IsomorphicQ[Wheel[10], GraphJoin[Cycle[9], EmptyGraph[1]]]
Out[31]= True
```

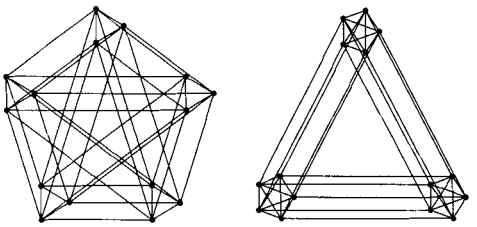
■ 6.1.6 Products of Graphs

The *product* $G_1 \times G_2$ of two graphs has a vertex set defined by the Cartesian product of the vertex sets of G_1 and G_2. There is an edge between (u_1, v_1) and (u_2, v_2) if $u_1 = u_2$ and v_1 is adjacent to v_2 in G_2 or $v_1 = v_2$ and u_1 is adjacent to u_2 in G_1. The intuition in taking the product is that all the vertices of one graph get replaced by instances of the other graph. Products of graphs have been studied extensively [IK00, Sab60]. *Combinatorica* provides a function `GraphProduct` that computes the product of a pair of graphs.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Graph products can be very interesting. The embedding of a product has been designed to show off its structure, and is formed by shrinking the first graph and translating it to the position of each vertex in the second graph. Reversing the order of the two arguments thus dramatically changes the appearance of the product, but the resulting graphs are isomorphic.

```
In[32]:= K3=CompleteGraph[3]; K5=CompleteGraph[5];
         ShowGraphArray[{g = GraphProduct[K3, K5],
         h = GraphProduct[K5, K3]}];
```



Since both graphs are always isomorphic, the product of two graphs is a commutative operation, even if our embeddings are different when the order of the arguments is changed.

```
In[34]:= IsomorphicQ[g,h]
Out[34]= True
```

Many of the properties of product graphs can be deduced from the corresponding properties of its factor graphs. For example, the product of two Hamiltonian graphs is Hamiltonian, and, similarly, if two graphs have Hamiltonian paths, then their product does too.

```
In[35]:= ShowGraph[g = GraphProduct[Wheel[6], Path[5]]]
```

A wheel and a path have Hamiltonian paths, and so does their product.

```
In[36]:= HamiltonianPath[g]
Out[36]= {1, 2, 3, 4, 5, 6, 12, 8, 9, 10, 11, 17, 16, 15,
    14, 18, 24, 20, 21, 22, 23, 29, 28, 27, 26, 30, 25, 19,
    13, 7}
```

Multiplication by K_1 is an identity operation, although there is no corresponding multiplicative inverse.

```
In[37]:= IdenticalQ[ GraphProduct[CompleteGraph[1], CompleteGraph[5]],
         CompleteGraph[5] ]
Out[37]= True
```

■ 6.1.7 Line Graphs

The *line graph* $L(G)$ of a graph G has a vertex of $L(G)$ associated with each edge of G and an edge of $L(G)$ if and only if the two edges of G share a common vertex. Line graphs are a special type of intersection graph, where each vertex represents a set of size 2 and each edge connects two sets with a nonempty intersection.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

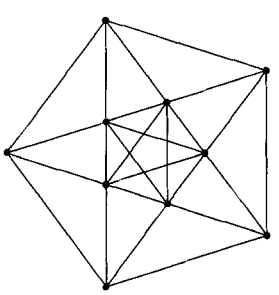
The obvious algorithm for constructing the line graph involves iterating through each pair of edges to decide whether they share a vertex between them, but the running time of this algorithm is quadratic in the number of edges of the input graph. However, the number of edges in $L(G)$ may be far smaller than the square of the number of edges in G. A more efficient algorithm is the following. First, lexicographically sort the edges and split them into groups by their first endpoint. Each group of edges shares their first endpoint and therefore form a clique in the line graph. The "helper" function, shown below, `GroupEdgePositions` computes the groupings of edges. Each edge grouping yields a clique, and the union of cliques is the line graph. The running time of this algorithm is proportional to the number of edges in the line graph plus the time to sort the edges in the input graph.

```
GroupEdgePositions[e_List, n_Integer] :=
      Map[Map[Last, #]&,
         Split[Union[Transpose[{e, Range[Length[e]]}], Table[{ {i, 0}, 0}, {i, n}]],
              (#1[[1,1]]=== #2[[1,1]])&
         ]
      ]

LineGraph[g_Graph] :=
      Module[{e = Sort[Edges[g]], ef, eb, c,
            v = Vertices[g]},
           ef = GroupEdgePositions[e, V[g]];
           eb = GroupEdgePositions[ Map[Reverse,e], V[g]];
           c  = Table[Rest[Union[ef[[i]], eb[[i]]]], {i, V[g]}];
           Graph[Union[
                 Flatten[Map[Table[{{#[[i]], #[[j]]}}, {i, Length[#]-1}, {j, i+1, Length[#]}]&, c], 2]
               ],
               Map[({(v[[ #[[1]] ]] + v[[ #[[2]] ]]) / 2})&, e]
           ]
      ]
```

Constructing Line Graphs

The line graph of a graph with n vertices and m edges contains m vertices and $\frac{1}{2}\sum_{i=1}^{n} d_i^2 - m$ edges. Thus the number of vertices of the line graph of K_n grows quadratically in n. Proofs of this and most of the results we cite on line graphs appear in [Har69]. The coordinates of each vertex in this embedding of $L(G)$ are the averages of the coordinates of the vertices associated with the original edge in G.

In[38]:= **ShowGraph[LineGraph[CompleteGraph[5]]];**



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Although some of the edges are ambiguous in the previous embedding, $L(K_5)$ is indeed a 6-regular graph.

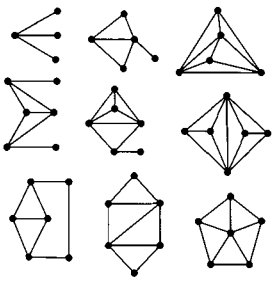
```
In[39]:= DegreeSequence[ LineGraph[CompleteGraph[5]] ]
Out[39]= {6, 6, 6, 6, 6, 6, 6, 6, 6, 6}
```

The cycle graph is the only connected graph that is isomorphic to its line graph.

```
In[40]:= IsomorphicQ[Cycle[10], LineGraph[Cycle[10]]]
Out[40]= True
```

No analogous `FromLineGraph` function is given, because not all graphs represent line graphs of other graphs. An example is the claw $K_{1,3}$. A structural characterization of line graphs is given in [vRW65] and refined by Beineke [Bei68], who showed that a graph is a line graph if and only if it does not contain any of these graphs as induced subgraphs.

```
In[41]:= ShowGraph[NonLineGraphs];
```

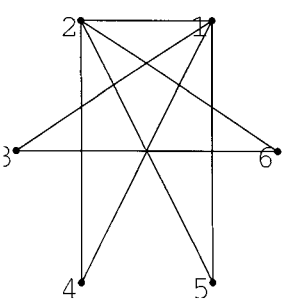


In introducing line graphs, Whitney [Whi32] showed that, with the exception of K_3 and $K_{1,3}$, any two connected graphs with isomorphic line graphs are isomorphic.

```
In[42]:= IsomorphicQ[LineGraph[CompleteGraph[3]],
         LineGraph[CompleteGraph[1,3]] ]
Out[42]= True
```

The *degree sequence* of a graph is the sequence of degrees of the vertices sorted in nonincreasing order. `RealizeDegreeSequence` takes a sequence of integers and constructs a random graph with this degree sequence. Here we get a 6-vertex graph of even degree. Such a graph has an Eulerian cycle and is said to be *Eulerian*.

```
In[43]:= ShowGraph[g = RealizeDegreeSequence[{4,4,2,2,2,2}],
         VertexNumber->True, TextStyle->{FontSize->12}];
```



An *Eulerian cycle* is a tour that visits every edge of the graph exactly once.

```
In[44]:= EulerianCycle[g]
Out[44]= {2, 4, 1, 3, 6, 2, 5, 1, 2}
```

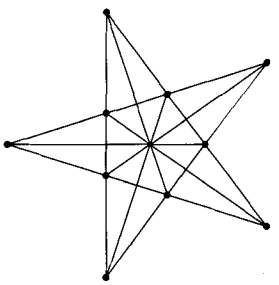
https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The line graph of an Eulerian graph is both Eulerian and Hamiltonian, while the line graph of a Hamiltonian graph is always Hamiltonian. Further results on cycles in line graphs appear in [Cha68, HNW65].

```
In[45]:= h = LineGraph[g];
         EulerianQ[h] && HamiltonianQ[h]
Out[46]= True
```

The complement of the line graph of K_5 is the Petersen graph. This is another example of an attractive but potentially misleading embedding. K_5 has 10 edges, so why does this graph have *11* vertices? The centermost "vertex" is just the intersection of five edges. Alternate embeddings of the Peterson graph appear in Section 5.1.4.

```
In[47]:= ShowGraph[ GraphComplement[ LineGraph[CompleteGraph[5]] ] ];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

6.2 Regular Structures

Many classes of graphs are defined by very regular structures that often suggests a natural embedding. They are typically parametrized by the number of vertices and occasionally edges.

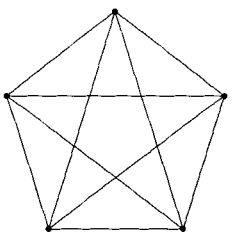
We have seen that many regular structures can be formulated using such operations as join and product. Here we give more efficient, special-purpose constructions for several classes of graphs.

6.2.1 Complete Graphs

A *complete graph* contains all possible edges. The complete undirected n-vertex graph K_n thus contains $\binom{n}{2}$ undirected edges, while the directed complete graph with n vertices contains $n(n-1)$ edges. Given a positive integer n, the function `CompleteGraph` produces K_n. The function for constructing complete graphs uses a circular embedding for the vertices.

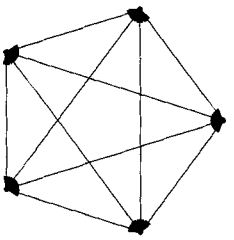
The complete graph on five vertices K_5 is famous as being the smallest nonplanar graph. Rotated appropriately, it becomes a supposed Satanic symbol, the pentagram.

```
In[48]:= ShowGraph[ RotateVertices[ CompleteGraph[5], Pi/10] ];
```



Using the option `Type -> Directed` produces the directed complete graph.

```
In[49]:= ShowGraph[CompleteGraph[5, Type -> Directed]];
```



The graph with no vertices and edges is called the *null graph* and has been studied extensively [HR73].

```
In[50]:= CompleteGraph[0]
Out[50]= -Graph:<0, 0, Undirected>-
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

A generalization of the null graph is the *empty graph* on n vertices, the complement of K_n.

```
In[51]:= EmptyGraph[10]
Out[51]= -Graph:<0, 10, Undirected>-
```

■ 6.2.2 Circulant Graphs

The *circulant* graph $C_n(n_1, n_2, ..., n_k)$ has n vertices with each v_i adjacent to each vertex $v_{i\pm n_j}$ mod n, where $n_1 < n_2 < \cdots < n_k < (n+1)/2$ [BH90]. Thus circulant graphs include complete graphs and cycles as special cases.

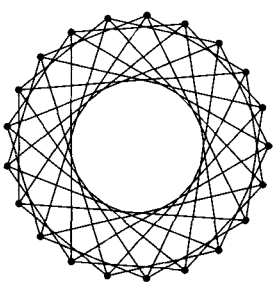
It is useful to think of circulant graphs as those whose adjacency matrix can be constructed by rotating a vector n times. However, to take advantage of their possible sparsity, we do not construct circulant graphs in this way.

```
CirculantGraph[n_Integer?Positive, l_Integer] := CirculantGraph[n, {l}]

CirculantGraph[n_Integer?Positive, l:{_Integer...}] :=
      Graph[Union[
              Flatten[Table[Map[{Sort[{i, Mod[i+#, n]}]+1}&, l], {i,0,n-1}], 1],
              Flatten[Table[Map[{Sort[{i, Mod[i-#, n]}]+1}&, l], {i,0,n-1}], 1]
           ],
           CircularEmbedding[n]
      ]
```

Constructing Circulant Graphs

Here is a circulant graph with 21 vertices, with each vertex connected to a vertex 3 away and a vertex 7 away on each side. The connections to vertices 3 away create three cycles of length 7 each, and the connections to vertices 7 away create seven cycles of length 3 each.

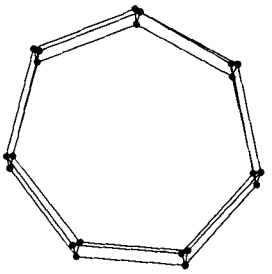
```
In[52]:= ShowGraph[g = CirculantGraph[21, {3, 7}]];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The spring embedding of the above graph makes clear this cycle structure. In fact, this circulant graph can be alternatively viewed as the product of a 3-cycle and a 7-cycle.

```
In[53]:= ShowGraph[SpringEmbedding[g, 30]];
```



Constructing a 21-vertex circulant graph by connecting to vertices 7 away leads to seven connected components, each a 3-cycle.

```
In[54]:= ConnectedComponents[ CirculantGraph[21, {7}] ]
Out[54]= {{1, 8, 15}, {2, 9, 16}, {3, 10, 17}, {4, 11, 18},
    {5, 12, 19}, {6, 13, 20}, {7, 14, 21}}
```

This is an example of an exciting general property.

```
In[55]:= g = CirculantGraph[15, {3, 5}];
         h = GraphProduct[Cycle[3],Cycle[5]];
         Isomorphism[g,h]
Out[57]= {1, 8, 15, 4, 11, 3, 7, 14, 6, 10, 2, 9, 13, 5, 12}
```

■ 6.2.3 Complete k-Partite Graphs

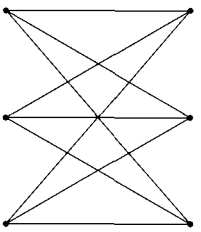
A graph is *bipartite* when its vertices can be partitioned into sets A and B such that every edge connects a vertex in A to a vertex in B. This notion can be generalized into *k-partite* graphs, whose vertices can be partitioned into k subsets such that no two vertices in the same subset are connected by an edge. A *complete* k-partite graph is a k-partite graph with every possible edge. Complete k-partite graphs are parametrized by the number of vertices in each of the k subsets. *Combinatorica* provides a function `CompleteKPartiteGraph` that takes as input the size of each of the k parts and returns the corresponding complete k-partite graph.

The standard way to draw a k-partite graph is as a leveled embedding in which vertices are partitioned into equally spaced stages, with the vertices of each stage drawn in a vertical line.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

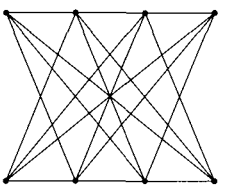
The most famous bipartite graph is $K_{3,3}$, the "other" smallest nonplanar graph, although $K_{18,18}$ plays an important role in the novel *Foucault's Pendulum* [Eco89]. A ranked embedding shows the structure of the graph since all edges are between vertices of different ranks.

```
In[58]:= ShowGraph[ CompleteKPartiteGraph[3,3] ];
```



We can construct complete k-partite graphs for any k. These graphs can get very dense when there are many stages. Whenever three vertices are collinear, there might be edges that overlap. This is unfortunate but costly to prevent.

```
In[59]:= ShowGraph[ CompleteKPartiteGraph[2,2,2,2] ];
```



The complete graph K_n can be defined as $K_{1,1,\ldots,1}$.

```
In[60]:= IsomorphicQ[CompleteGraph[5],
         CompleteKPartiteGraph[1,1,1,1,1]]
Out[60]= True
```

A special case of a complete k-partite graph is the *Turán graph*, which provided the answer to the first problem in what is now known as extremal graph theory [Tur41]. An *extremal graph* $Ex(n, G)$ is the largest graph of order n that does not contain G as a subgraph. Turán was interested in finding the extremal graph that does not contain K_p as a subgraph. Since Turán's paper, extremal graph theory has grown to have a very large literature [Bol78].

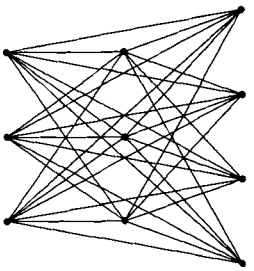
```
Turan[n_Integer, 2] := GraphUnion[n, CompleteGraph[1]] /; (n > 0)
Turan[n_Integer,p_Integer] :=
      Module[{k = Floor[ n / (p-1) ], r},
            r = n - k (p-1);
            Apply[CompleteGraph, Join[Table[k,{p-1-r}], Table[k+1,{r}]]]
      ] /; (n >= p) && (p > 2)
Turan[n_Integer, p_Integer] := CompleteGraph[n] /; (n < p) && (p > 2)
```

Constructing the Turán Graph

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Since the Turán graph is $(p-1)$-partite, it cannot contain K_p as a subgraph. The idea behind the construction is to balance the size of each stage as evenly as possible, maximizing the number of edges between every pair of stages.

```
In[61]:= ShowGraph[Turan[10,4]];
```

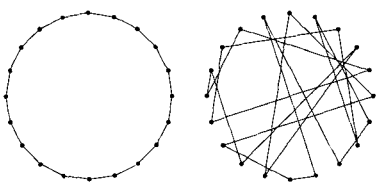


■ 6.2.4 Cycles, Stars, and Wheels

The *cycle* C_n is the connected 2-regular graph of order n. Special cases of interest include the *triangle* K_3 and the *square* C_4. Cycles are a special case of circulant graphs.

Here we show two different embeddings of C_{20}. Both embeddings are circular, but there are no edge crossings in the default embedding. The minimum number of swaps between vertices in a random circular embedding of a cycle to get it into its proper configuration is an interesting combinatorial problem. This is related to the *Bruhat order* of a group [BW82, Sta86].

```
In[62]:= ShowGraphArray[{g = Cycle[20], h = PermuteSubgraph[Cycle[20],
           RandomPermutation[20]]}];
```



A *star* is a tree with one vertex of degree $n-1$. Since a star is a tree, it is acyclic. An interesting property of the star is that the distance between any two vertices is at most two, despite the fact that it contains the minimum number of edges to be connected. This makes it a good topology for computer networks because it minimizes communication time between nodes, at least until the center node goes down.

```
Star[n_Integer?Positive] :=
      Graph[Table[{{i, n}}, {i, n-1}],
            Append[CircularEmbedding[n-1], {{0, 0}}]
      ]
```

Constructing a Star

In this construction, the center of the star is the nth vertex.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The complete bipartite graph $K_{1,n-1}$ is a star on n vertices.

```
In[63]:= Isomorphism[ CompleteKPartiteGraph[1,5], Star[6] ]
Out[63]= {6, 1, 2, 3, 4, 5}
```

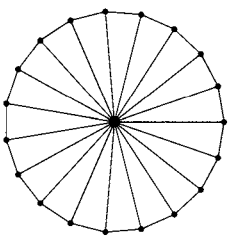
A *wheel* is the graph obtained by joining a single isolated vertex K_1 to each vertex of a cycle C_{n-1}. The resulting edges that form the star are, logically enough, called the *spokes* of the wheel.

```
Wheel[n_Integer] :=
      Graph[Join[Table[{{i, n}}, {i, n-1}], Table[{{i, i+1}}, {i, n-2}],
                 {{{1, n-1}}}
            ],
            Append[CircularEmbedding[n-1], {{0, 0}}]
      ] /; (n >= 3)
```

Constructing a Wheel

The *dual* of a planar embedding of a graph G is a graph with a vertex for each region of G, with edges if the corresponding regions are adjacent. The dual graph of a wheel is a wheel.

```
In[64]:= ShowGraph[ Wheel[20] ];
```



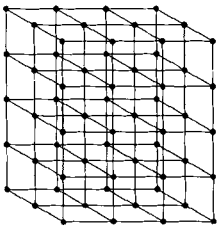
■ 6.2.5 Grid Graphs

Anyone who has used a piece of graph paper is familiar with *grid graphs*. The vertices in an $m \times n$ grid graph correspond to ordered pairs (i,j), where i and j are integers, $1 \le i \le m$, and $1 \le j \le n$. Every edge connects pairs (i,j) and (i',j'), where $|i - i'| + |j - j'| = 1$. Counting the number of distinct paths between two points in a grid graph is a classic application of binomial coefficients, discussed in Section 8.1.4. An $m \times n$ grid graph can be constructed by computing the product of paths of orders m and n. Grid graphs can be generalized to higher dimensions in a natural way. *Combinatorica* provides a function `GridGraph` that takes as input two or three positive integers and returns a two-dimensional or a three-dimensional grid graph that is appropriately embedded.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

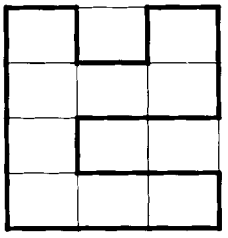
Combinatorica provides a constructor for three-dimensional grid graphs. A $p \times q \times r$ grid graph can be constructed by computing the product of three paths.

```
In[65]:= ShowGraph[g = GridGraph[4,5,3]];
```



Grid graphs are Hamiltonian if the number of either rows or columns is even. Such a cycle can be constructed by starting in the lower left-hand corner, going all way to the right, and then zig-zagging up and down until reaching the original corner. Other Hamiltonian cycles, like this one, do not follow this pattern.

```
In[66]:= ShowGraph[Highlight[g = GridGraph[4, 5],
           {Partition[HamiltonianCycle[g], 2, 1]}]];
```



Even this small graph has many Hamiltonian cycles.

```
In[67]:= Length[HamiltonianCycle[g, All]]
Out[67]= 28
```

Grid graphs are bipartite, for the vertices can be partitioned like the squares on a chessboard.

```
In[68]:= TwoColoring[ GridGraph[4,5] ]
Out[68]= {1, 2, 1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1,
    1, 2, 1, 2}
```

■ 6.2.6 Interconnection Networks

A parallel computer consists of some number of processors connected by an *interconnection network*. Designers of parallel computers typically seek interconnection networks that have few connections but have a small diameter and in which routing of information is easy. The *hypercube* is one of the most versatile and efficient networks for parallel computation [Lei92]. One drawback of the hypercube is that the number of connections to each processor grows logarithmically with the size of the network. Several *hypercubic networks* have been proposed to overcome this problem, including the butterfly, shuffle exchange, and the De Bruijn graphs. In this section, we describe graph-theoretic properties of these networks and provide constructors for them.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Hypercubes

An *n-dimensional hypercube* or an *n-cube* is defined as the product of K_2 and an $(n-1)$-dimensional hypercube. This recursive definition can be used to construct them.

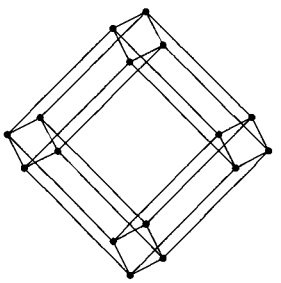
```
Hypercube[n_Integer] := Hypercube1[n]

Hypercube1[0] := CompleteGraph[1]
Hypercube1[1] := Path[2]
Hypercube1[2] := Cycle[4]

Hypercube1[n_Integer] := Hypercube1[n] =
      GraphProduct[
            RotateVertices[Hypercube1[Floor[n/2]], 2Pi/5],
            Hypercube1[Ceiling[n/2]]
      ]
```

Constructing Hypercubes

All hypercubes are Hamiltonian, as proven in Section 2.3.2 when we constructed the binary-reflected Gray code. The vertices of an n-dimensional hypercube can be labeled with length n binary strings such that adjacent vertices differ in exactly 1 bit. Such labelings can be constructed recursively. Label the two vertices of a one-dimensional hypercube 0 and 1. Partition an n-dimensional hypercube into two $(n-1)$-dimensional hypercubes and prepend 0 to the vertex labels of one $(n-1)$-dimensional hypercube and 1 to the vertex labels of the other.

Here is a four-dimensional hypercube. It can be viewed as two three-dimensional cubes connected in a symmetric way. An n-dimensional hypercube has 2^n vertices, with each vertex having degree n, for an total of $n2^{n-1}$ edges.

In[69]:= **ShowGraph[g = Hypercube[4]];**



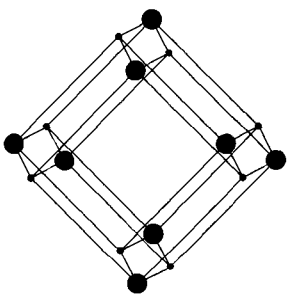
https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

This shows the distribution of vertices according to their distance from a particular vertex in a d-dimensional hypercube. Does it look familiar? These numbers are binomial coefficients, reinforcing the connection between subsets, binary strings, and hypercubes.

```
In[70]:= Table[ Distribution[ First[ AllPairsShortestPath[Hypercube[i]] ] ],
         {i,0,6}] // TableForm
Out[70]//TableForm= 1
                    1  1
                    1  2  1
                    1  3  3  1
                    1  4  6  4  1
                    1  5  10 10 5  1
                    1  6  15 20 15 6  1
```

All n-cubes are bipartite, as can be seen from the recursive construction.

```
In[71]:= c = TwoColoring[g];
         v1 = Flatten[Position[c, 1]]; v2 = Flatten[Position[c, 2]];
         ShowGraph[ Highlight[g, {v1}]]
```



Butterfly Graphs

The *r-dimensional butterfly graph* has $(r+1)2^r$ vertices and $r2^{r+1}$ edges. The vertices correspond to pairs (w, i), where i is the *level* or *dimension* of the vertex ($0 \le i \le r$) and w is an r-bit binary number that denotes the *row* of the vertex. Two vertices (w, i) and $(w', i+1)$ are connected by an edge if and only if either (i) w and w' are identical, or (ii) w and w' differ in precisely the $(i+1)$th bit.

```
Options[ButterflyGraph] = {VertexLabel->False}

ButterflyGraph[n_Integer?Positive, opts___?OptionQ] :=
      Module[{v = Map[Flatten, CartesianProduct[Strings[{0, 1}, n], Range[0, n]]], label},
            label = VertexLabel /. Flatten[{opts, Options[ButterflyGraph]}];
            RankedEmbedding[
               MakeUndirected[
                  MakeGraph[v,
                           (#1[[n+1]]+1 == #2[[n+1]]) &&
                           (#1[[Range[#2[[n+1]]-1]]] == #2[[Range[#2[[n+1]]-1]]]) &&
                           (#1[[Range[#2[[n+1]]+1, n]]] == #2[[Range[#2[[n+1]]+1, n]]])&,
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

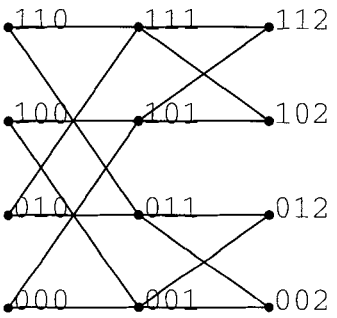
```
                    VertexLabel -> label
                ]
            ],
            Flatten[Position[v, {__, 0}]]
        ]
    ]
```

Constructing a Butterfly Graph

To construct a butterfly graph, we use `CartesianProduct` to obtain the set of all pairs (w, i) and then `MakeGraph` to construct the edges from the pairs. `MakeGraph` takes as input the set of vertices (w, i) and a Boolean predicate that indicates which pairs of vertices are to be connected by edges. Finally, a ranked embedding of the graph is constructed, with rows representing vertices that have identical w and columns representing vertices that have identical levels.

This is the two-dimensional butterfly graph. To produce cleaner labels, i is appended to w in the label for vertex (w, i). Edges that connect vertices with the same binary string are called *straight* edges. The remaining edges are called *cross* edges and go between level i and level $i + 1$ vertices that differ in the $(i + 1)$th bit.

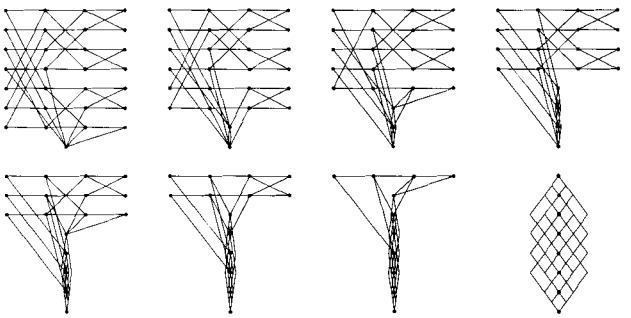
```
In[74]:= ShowGraph[g = ButterflyGraph[2, VertexLabel -> True],
         TextStyle->{FontSize->12}, PlotRange->0.2];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

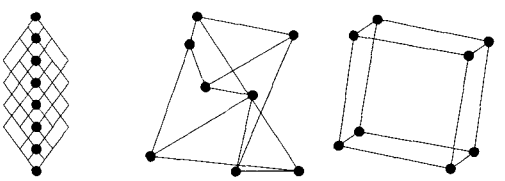
Here we show a series of graphs obtained by contracting rows of vertices in a three-dimensional butterfly. The bottommost row is contracted first, then the row above it, and so on. The contracted vertices end up on a vertical line with multiple edges between pairs of these. At the end we have eight vertices on a vertical line with the multiple edges making a nice pattern.

```
In[75]:= g = Contract[ButterflyGraph[3], {1, 2, 3, 4}];
         ShowGraphArray[Partition[l = NestList[Contract[#,
         {1, 2, 3, 4}] &, g, 7], 4]];
```



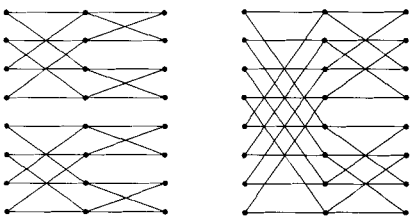
We start with the final 8-vertex graph obtained by repeatedly contracting the butterfly graph, make it simple by getting rid of multiple edges, and finally give it a new embedding to reveal it as a three-dimensional hypercube, which is just a "folded-up" butterfly. If all vertices from the same level in an r-dimensional butterfly are contracted to a single vertex, we get an r-dimensional hypercube.

```
In[77]:= ShowGraphArray[{g = l[[8]], h = RandomVertices[MakeSimple[g]],
                SpringEmbedding[h,200] }, VertexStyle->Disk[0.06]];
```



The butterfly graph also has a beautiful recursive structure. Deleting the level 0 vertices of an r-dimensional butterfly leaves two $(r-1)$-dimensional butterflies, as shown in the graph on the left. Deleting the level r vertices *also* gives two $(r-1)$-dimensional butterflies. This is shown on the right, though it is less obvious that this graph has two connected components.

```
In[78]:= ShowGraphArray[{DeleteVertices[g = ButterflyGraph[3],
         Table[i, {i, 1, V[g], 4}]],
         h = DeleteVertices[g, Table[i, {i, 4, V[g], 4}]]}]
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The graph obtained by deleting the level r vertices from an r-dimensional butterfly indeed contains two equal-sized connected components.

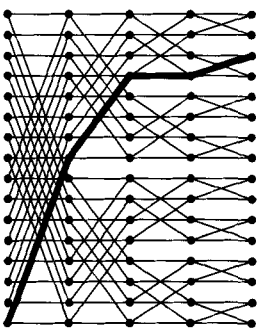
```
In[79]:= w = ConnectedComponents[h]
Out[79]= {{1, 2, 3, 7, 8, 9, 13, 14, 15, 19, 20, 21},
    {4, 5, 6, 10, 11, 12, 16, 17, 18, 22, 23, 24}}
```

Further, each of these components is isomorphic to the two-dimensional butterfly.

```
In[80]:= {IsomorphicQ[InduceSubgraph[h, w[[1]]], ButterflyGraph[2]],
          IsomorphicQ[InduceSubgraph[h, w[[2]]], ButterflyGraph[2]]}
Out[80]= {True, True}
```

There is a unique shortest path between each level 0 vertex w and each level r vertex w'. The path traverses each level exactly once, using the cross edge from level i to level $(i+1)$ if and only if w and w' differ in the $(i+1)$st bit. A consequence of this is that N-vertex butterflies have diameter $O(\log N)$.

```
In[81]:= ShowGraph[Highlight[g = ButterflyGraph[4],
         {Partition[ShortestPath[g, 1, 70], 2, 1]} ]];
```



Shuffle-Exchange Graphs

The r-dimensional *shuffle-exchange* graph has $N = 2^r$ vertices and $3 \cdot 2^{r-1}$ edges. Each vertex corresponds to a unique r-bit binary string, and two vertices u and v are connected by an edge if (a) u and v differ precisely in the last bit or (b) u is a left or right cyclic shift of v. If u and v differ in the last bit, the edge is called an *exchange edge*; otherwise the edge is called a *shuffle edge*.

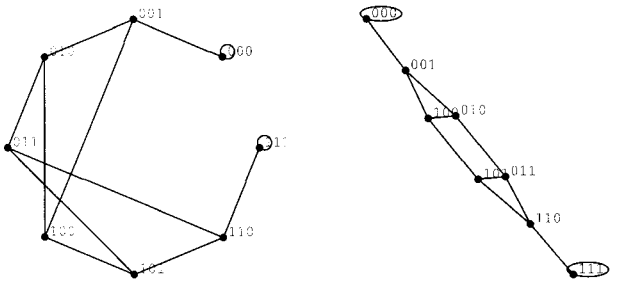
```
Options[ShuffleExchangeGraph] = {VertexLabel -> False}

ShuffleExchangeGraph[n_Integer?Positive, opts___?OptionQ] :=
      Module[{label},
            label = VertexLabel /. Flatten[{opts, Options[MakeGraph]}];
            MakeGraph[Strings[{0, 1}, n],
                  (Last[#1] != Last[#2]) && (Take[#1,(n-1)]==Take[#2,(n-1)]) ||
                  (RotateRight[#1,1] == #2) || (RotateLeft[#1, 1] == #2)&,
                  Type -> Undirected, VertexLabel -> label
            ]
      ]
```

Constructing a Shuffle-Exchange Graph

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The three-dimensional shuffle-exchange graph is shown here to reveal its recursive structure. Consider the vertex 011. It is connected by an exchange edge to 010 and by shuffle edges to 101 and 110. A few applications of `SpringEmbedding` produces the drawing that is more often seen in textbooks.

```
In[82]:= ShowGraphArray[{g = ShuffleExchangeGraph[3, VertexLabel -> True],
           SpringEmbedding[g, 100]}, PlotRange -> 0.1];
```



Here we report a shortest path between a pair of randomly chosen vertices in a five-dimensional shuffle-exchange graph. Note that every vertex is obtained from the previous one by either a shuffle operation or an exchange operation.

```
In[83]:= g = ShuffleExchangeGraph[5, VertexLabel -> True];
         s = Random[Integer, {1, 32}]; t = Random[Integer, {1, 32}];
         GetVertexLabels[g][[ShortestPath[g, s, t]]]
Out[85]= {01111, 01110, 00111, 00110, 00011, 10001}
```

Any r-bit binary string u can be converted to any other r-bit binary string v by $r-1$ shuffles and at most r exchanges. This implies that an r-dimensional shuffle-exchange graph has diameter $2r-1 = 2\log N - 1$. Going from $00\ldots0$ to $11\ldots1$ requires exactly r exchanges for a total of $2r-1$ operations.

```
In[86]:= Table[Diameter[ShuffleExchangeGraph[i]], {i, 1, 7}]
Out[86]= {1, 3, 5, 7, 9, 11, 13}
```

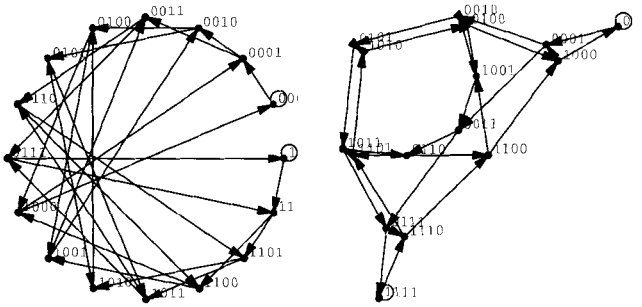
De Bruijn Graphs

The *r-dimensional De Bruijn graph* consists of 2^r vertices and 2^{r+1} directed edges. Each vertex corresponds to a unique r-bit binary string. There is a directed edge from each node $u_1u_2\ldots u_r$ to $u_2u_3\ldots u_r0$ and to $u_2u_3\ldots u_r1$. Each vertex in a De Bruijn graph has in-degree 2 and out-degree 2. *Combinatorica* provides a function `DeBruijnGraph` that constructs De Bruijn graphs.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

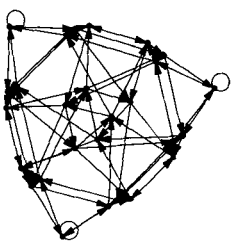
This shows the three-dimensional De Bruijn graph. Each vertex has in-degree 2 and out-degree 2, assuming that the self-loops at vertices 000 and 111 each contribute one in-coming and one out-going edge. Applying `SpringEmbedding` to the De Bruijn graph reveals the symmetry in its structure.

```
In[87]:= ShowGraphArray[{g = DeBruijnGraph[{0, 1}, 4, VertexLabel -> True],
          SpringEmbedding[g, 100]}];
```



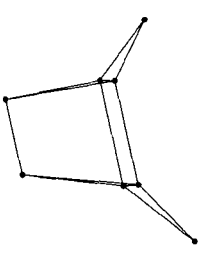
The function `DeBruijnGraph` also constructs graphs over alphabets with more than two elements. Here the nine vertices correspond to all ternary strings of length 2. For each string x and each symbol α in the alphabet, there is an edge going from x to the string obtained by shifting x to the left one position and inserting α into the last position. In the case of the ternary De Bruijn graph, this means that each vertex has in-degree and out-degree equal to 3.

```
In[88]:= ShowGraph[SpringEmbedding[ DeBruijnGraph[3, 3] ]];
```



The important connection between shuffle-exchange graphs and De Bruijn graphs is as follows. Take an $(r+1)$-dimensional shuffle-exchange graph and "contract" out all the exchange edges from the graph. In other words, replace every pair of vertices $u_1u_2\ldots u_r0$ and $u_1u_2\ldots u_r1$ by $u_1u_2\ldots u_r$. This leaves only the shuffle edges.

```
In[89]:= g = ShuffleExchangeGraph[4]; Do[g = Contract[g, {1, 2}], {8}];
          g = MakeSimple[g]; ShowGraph[Nest[SpringEmbedding, g, 10]];
```



The resulting graph is isomorphic to the undirected version of the De Bruijn graph.

```
In[91]:= IsomorphicQ[g, MakeSimple[DeBruijnGraph[2, 3]]]
Out[91]= True
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

6.3 Trees

A *tree* is a connected graph with no cycles. Trees are the simplest interesting class of graphs, so "Can you prove it for trees?" should be the first question asked after formulating a new graph-theoretic conjecture. In this section we first discuss the problem of enumerating labeled trees. With most enumeration problems, counting the number of unlabeled objects is harder than counting the number of labeled ones, and so it is with trees. Algorithms for the systematic generation of free and rooted trees appear in [NW78, Wil89].

6.3.1 Labeled Trees

One of the first theorems in graphical enumeration was Cayley's proof [Cay89] that there are n^{n-2} distinct labeled trees on n vertices. Prüfer [Pr8] established a bijection between such trees and strings of $n-2$ integers between 1 and n, providing a constructive proof of Cayley's result. This bijection can then be exploited to give algorithms for systematically and randomly generating labeled trees.

The key to Prüfer's bijection is the observation that for any tree there are always at least two *leaves*, that is, vertices of degree 1. Start with an n-vertex tree T, whose vertices are labeled 1 through n. Let u be the leaf with the smallest label and let v be the neighbor of u. Note that u and v are uniquely defined. We now let v be the first symbol in our string, or Prüfer code. After deleting vertex u we have a tree on $n-1$ vertices, and repeating this operation until only one edge is left gives us $n-2$ integers between 1 and n.

```
LabeledTreeToCode[g_Graph] :=
      Module[{e=ToAdjacencyLists[g],i,code},
            Table [
                  {i} = First[ Position[ Map[Length,e], 1 ] ];
                  code = e[[i,1]];
                  e[[code]] = Complement[ e[[code]], {i} ];
                  e[[i]] = {};
                  code,
                  {V[g]-2}
            ]
      ]
```

Constructing a Prüfer Code from a Labeled Tree

To reconstruct T from its Prüfer code, we observe that a particular vertex appears in the code exactly one time less than its degree in T. Thus we can compute the degree sequence of T and thereby identify the lowest labeled degree-1 vertex in the tree. Since the first symbol in the code is the vertex it is incident upon, we have determined the first edge and, by induction, the entire tree.

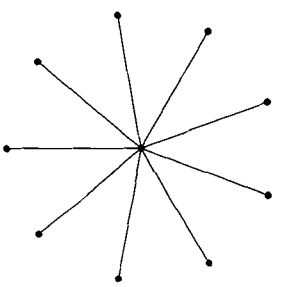
https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

```
CodeToLabeledTree[l_List] :=
      Module[{m=Range[Length[l]+2],x,i},
            FromUnorderedPairs[
                  Append[
                        Table[
                              x = Min[Complement[m,Drop[l,i-1]]];
                              m = Complement[m,{x}];
                              Sort[{x,l[[i]]}],
                              {i,Length[l]}
                        ],
                        Sort[m]
                  ]
            ]
      ] /; (Complement[l, Range[Length[l]+2]] == {})
```

Constructing a Labeled Tree from its Code

A star contains $n-1$ vertices of degree 1, all incident on one center. Since the degree of a particular vertex is one more than its frequency in the Prüfer code, the ith labeled star is defined by a code of $n-2$ i's.

```
In[92]:= ShowGraph[ RadialEmbedding[
         CodeToLabeledTree[{10,10,10,10,10,10,10,10}] ] ];
```



Since there is a bijection between trees and codes, composing the two functions gives an identity operation.

```
In[93]:= LabeledTreeToCode[ CodeToLabeledTree[{3,3,3,2,3}] ]
Out[93]= {3, 3, 3, 2, 3}
```

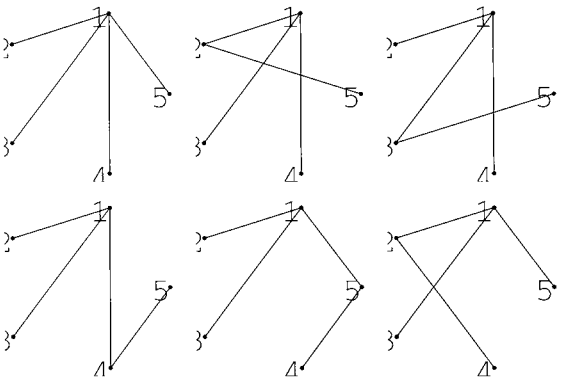
Each labeled path on n vertices is represented by a pair of permutations of length n, for a total of $n!/2$ distinct labeled paths. The Prüfer codes of paths are exactly the sequences of $n-2$ distinct integers, since the two vertices of degree 1 do not appear in the code.

```
In[94]:= LabeledTreeToCode[Path[10]]
Out[94]= {2, 3, 4, 5, 6, 7, 8, 9}
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Enumerating all labeled trees is a matter of enumerating Prüfer codes and applying CodeToLabeledTree. Here we use the function Strings to enumerate Prüfer codes for 5-vertex trees and then convert the first six codes into labeled trees. There are only two distinct unlabeled trees in this list because the last five trees are isomorphic.

```
In[95]:= s = Strings[Range[5], 3];
         ShowGraphArray[Partition[Map[CodeToLabeledTree,
         s[[Range[6]]]], 3], VertexNumber -> True,
         TextStyle -> {FontSize -> 12}];
```



Cayley's result was the first of many in graphical enumeration [HP73], counting how many distinct graphs of a given type exist. An interesting table of the number of a dozen different types of graphs on up to eight vertices appears in [RW97, Wil85].

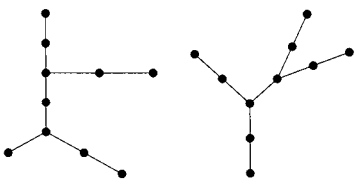
The bijection between Prüfer codes and labeled trees yields an algorithm for selecting a random labeled tree. Simply generate a random Prüfer code and convert it into the tree.

```
RandomTree[1] := Graph[{}, {{{0, 0}}}]
RandomTree[n_Integer?Positive] :=
      RadialEmbedding[CodeToLabeledTree[Table[Random[Integer,{1,n}], {n-2}] ], 1]
```

Selecting a Random Labeled Tree

Here are four randomly selected 10-vertex trees. RandomTree uses a radial embedding as the default embedding for the constructed tree.

```
In[97]:= ShowGraphArray[Table[RandomTree[10], {2}],
         VertexStyle -> Disk[0.05]];
```



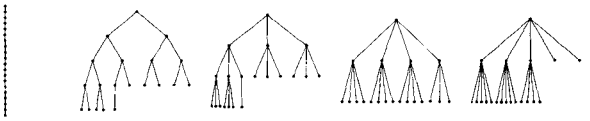
https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

■ 6.3.2 Complete Trees

In a rooted tree, the distance between a vertex v and the root is called the *depth* of v. The maximum depth of any vertex in a rooted tree is the *height* of the tree. A *k-ary tree* is a rooted tree in which each vertex has at most k children. A *complete k-ary tree* is a k-ary tree in which all vertices at depth $h-1$ or less have exactly k children, where h is the height of the tree. With this definition, for fixed n and k, there can be several distinct n-vertex complete k-ary trees, depending on how the leaves of the tree are distributed. The function `CompleteKaryTree` takes as input n and k and produces a complete k-ary tree with n vertices in which leaves in the last level are all on the "left."

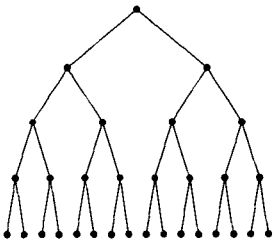
These are complete k-ary trees with 20 vertices for $1 \le k \le 5$. Notice that the tree gets shorter with increasing k. A special case of a complete k-ary tree is a complete binary tree, where $k = 2$.

```
In[98]:= ShowGraphArray[Table[CompleteKaryTree[20, i], {i,5}]];
```



A *complete binary tree* is a special case of complete k-ary trees for $k = 2$. `CompleteBinaryTree` produces complete binary trees.

```
In[99]:= ShowGraph[ CompleteBinaryTree[31] ]
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

6.4 Random Graphs

The easiest way to generate a graph is by tossing a coin for each edge to decide whether it should be included. The theory of *random graphs* considers questions of the type, "What density of edges in a graph is necessary, on average, to ensure that the graph has monotone graph property X?". A *monotone graph property* is an invariant that is preserved as more edges are added, such as whether the graph is connected or has a cycle. The theory of random graphs was introduced by Erdös and Renyi in a classic paper [ER60].

6.4.1 Constructing Random Graphs

The classical theory of random graphs deals with two primary models. In the first, a parameter p represents the probability that any given edge is in the graph. In other words, each edge is chosen *independently* with probability p to be in the graph. When $p = 1/2$, this model generates all labeled graphs with equal probability, since the edge probabilities are independent.

The simplest way to construct a random graph in this model is to build an adjacency matrix in which each entry is set to 1 with probability p. However, the running time of this algorithm is $\Theta(n^2)$, independent of p. When p is very small, the graph has few edges, so this is rather inefficient. We make the algorithm much faster by noting that the number of edges in the graph has a binomial distribution with $\binom{n}{2}$ trials, each with success of probability p. The *Mathematica* add-on package `Statistics`DiscreteDistributions`` contains a function `BinomialDistribution` that allows us to define a random variable d that has this distribution with the appropriate parameters. Picking a random value for this random variable gives us the number of edges the graph should have. We then use the *Combinatorica* function `RandomKSubset` to generate a random subset of size d of the set $\{1, 2, \ldots, \binom{n}{2}\}$, which gives us the indices of the edges in the graph. Finally, we turn these indices into actual edges by using the *Combinatorica* function `NthPair`.

```
RandomGraph[n_Integer?Positive, p_?NumericQ, opts___?OptionQ] :=
      Module[{type},
            type = Type /. Flatten[{opts, Options[RandomGraph]}];
            If[type === Directed, RDG[n, p], RG[n, p] ]
      ]

RG[n_, p_] :=
      Module[{d = BinomialDistribution[Binomial[n, 2], p]},
            Graph[Map[{NthPair[#]}&, RandomKSubset[Range[Binomial[n,2]], Random[d]]], CircularEmbedding[n]]
      ]
```

Constructing Model 1 Random Graphs

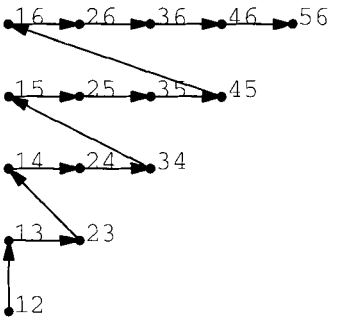
https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

NthPair is a function that unranks all unordered pairs of positive integers.

```
In[100]:= NthPair[1000000]
Out[100]= {1009, 1415}
```

The ordering that NthPair assumes is shown here as a directed path. From the picture, it is clear that if n is the rank of the unordered pair (i,j), then j is the largest integer such that $(j-1)(j-2)/2 \le n$ and $i = n - (j-1)(j-2)/2$. The code for NthPair solves this quadratic equation for j.

```
In[101]:= ShowGraph[ChangeVertices[MakeGraph[v = Map[NthPair, Range[15]],
            Position[v, #1][[1, 1]] + 1 == Position[v, #2][[1, 1]] &,
            VertexLabel -> True], v], TextStyle->{FontSize->10},
            PlotRange->0.2];
```

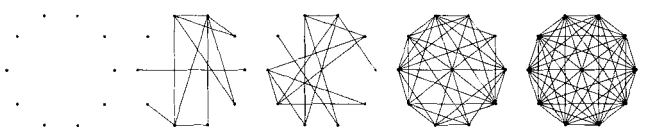



```
NthPair = Compile[{{n, _Integer}},
          Module[{j}, j = Ceiling[(1 + Sqrt[1 + 8n])/2]; {Round[n - (j-1)(j-2)/2], j}]
        ]
```

Unranking Unordered Pairs

This shows 10-vertex random graphs with increasing p, from 0 to 1. We start with an empty graph and end up with a complete graph as p increases.

```
In[102]:= ShowGraphArray[Table[RandomGraph[10, p], {p, 0, 1, .25}]];
```



The interesting issue in random graph theory is how many edges are needed before the graph can be expected to have some monotone graph property. Random graph theory is discussed in [Bol79, ES74, Pal85, JLR00]. Through the probabilistic model, it can be proven that almost all graphs are connected and almost all graphs are nonplanar.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Large model 1 random graphs with probability 1/2 seem likely to be connected, while those with probability $1/n$ seem disconnected. At what p does the transition from disconnected to connected take place? These kinds of questions are addressed by random graph theory.

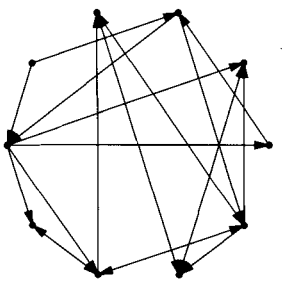
```
In[103]:= {Map[ConnectedQ, Table[RandomGraph[n, .5], {n, 100, 200, 15}]],
           Map[ConnectedQ, Table[RandomGraph[n, 1/n], {n, 100, 200, 15}]]} //
           TableForm
Out[103]//TableForm=
True    True    True    True    True    True    True
False   False   False   False   False   False   False
```

There are eight labeled graphs on three vertices. When the edge probability is 1/2, each is equally likely to occur. Larger complete catalogs of graphs appear in [RW97] and on the page maintained by Gordon Royle [Roy02].

```
In[104]:= Distribution[ Table[RandomGraph[3,0.5], {200}] ]
Out[104]= {25, 26, 23, 21, 27, 29, 22, 27}
```

Random directed graphs can be constructed in a similar manner by using the option `Type -> Directed`.

```
In[105]:= ShowGraph[ RandomGraph[10, .3, Type -> Directed]];
```



The second model generates random unlabeled graphs with exactly m edges, which is difficult to do both efficiently and correctly, since it implies describing the set of nonisomorphic graphs. Our algorithm produces random *labeled* graphs of the given size, by selecting m integers from 1 to $\binom{n}{2}$ and defining a bijection between such integers and ordered pairs. This does not result in a random unlabeled graph, as shown in the following example.

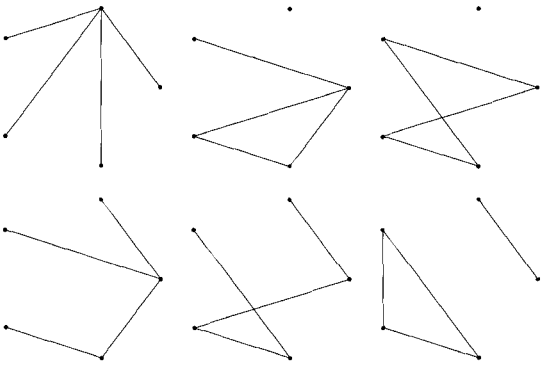
```
ExactRandomGraph[n_Integer,e_Integer] :=
      Graph[Map[{NthPair[#]}&, Take[ RandomPermutation[n(n-1)/2], e] ], CircularEmbedding[n]]
```

Constructing Random Labeled Graphs on m Edges

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

This shows the six unlabeled 5-vertex graphs with four edges. Of these, three are trees, one of which is the path of length 5.

```
In[106]:= ShowGraphArray[Partition[ListGraphs[5, 4], 3]];
```



Here we generate 1000 instances of 5-vertex random graphs via `ExactRandomGraph`, each containing four edges. Very few are identical, but...

```
In[107]:= Length[Union[l = Table[ExactRandomGraph[5, 4], {1000}]]]
Out[107]= 910
```

...many are isomorphic. More than a quarter of the random trees are 5-paths. If `ExactRandomGraph` had generated unlabeled graphs uniformly at random, this number would have been about one-sixth of the total number generated.

```
In[108]:= Count[Map[TreeIsomorphismQ[Path[5], #] &, l], True]
Out[108]= 272
```

Stars are extremely rare as random labeled trees. Why?

```
In[109]:= Count[Map[TreeIsomorphismQ[Star[5], #] &, l], True]
Out[109]= 37
```

■ 6.4.2 Realizing Degree Sequences

The *degree sequence* of an undirected graph is the number of edges incident on each vertex. Perhaps the most elementary theorem in graph theory is that the sum of the degree sequence for any graph must be even, or, equivalently, that any simple graph has an even number of odd-degree vertices. By convention, degree sequences are sorted into decreasing order. The degree of a vertex is sometimes called its *valency*, and the minimum and maximum degrees of graph G are denoted $\delta(G)$ and $\Delta(G)$, respectively.

A degree sequence is *graphic* if there exists a simple graph with that degree sequence. Erdös and Gallai [EG60] proved that a degree sequence is graphic if and only if the sequence observes the

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

following condition for each integer $r < n$:

$$\sum_{i=1}^{r} d_i \leq r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i).$$

The Erdös-Gallai condition also generalizes to directed graphs [Ful65, Rys57].

Alternatively, [Hak62, Hav55] proved that if a degree sequence is graphic, there exists a graph G where the vertex of highest degree is adjacent to the $\Delta(G)$ next highest degree vertices of G. This gives an inductive definition of a graphic degree sequence, as well as a construction of a graph that realizes it.

```
DegreeSequence[g_Graph] := Reverse[ Sort[ Degrees[g] ] ]
Degrees[g_Graph] := Map[Length, ToAdjacencyLists[g]]

GraphicQ[s_List] := False /; (Min[s] < 0) || (Max[s] >= Length[s])
GraphicQ[s_List] := (First[s] == 0) /; (Length[s] == 1)
GraphicQ[s_List] :=
      Module[{m,sorted = Reverse[Sort[s]]},
            m = First[sorted];
            GraphicQ[ Join[ Take[sorted,{2,m+1}]-1, Drop[sorted,m+1] ] ]
      ]
```

Identifying Graphic Degree Sequences

The degree sequence shows that K_{30} is a 29-regular graph.

```
In[110]:= DegreeSequence[CompleteGraph[30]]
Out[110]= {29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29,
     29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29,
     29, 29, 29, 29}
```

Degrees returns the degrees of vertices in vertex order, whereas DegreeSequence produces these in nonincreasing order.

```
In[111]:= g = RandomGraph[10, .5];
          {Degrees[g], DegreeSequence[g]} // ColumnForm
Out[112]= {3, 3, 3, 6, 5, 4, 3, 6, 5, 8}
          {8, 6, 6, 5, 5, 4, 3, 3, 3, 3}
```

By definition, the degree sequence of any graph is graphic.

```
In[113]:= GraphicQ[DegreeSequence[RandomGraph[10,1/2]]]
Out[113]= True
```

The diagonal of the square of the adjacency matrix of a simple graph gives the unsorted degree sequence of the graph. This is because the square of the adjacency matrix counts the number of walks of length 2 between any pair of vertices, and the only walks of length 2 from a vertex to itself are to an adjacent vertex and directly back again.

```
In[114]:= (g = RealizeDegreeSequence[{3, 2, 2, 1}];
          TableForm[ToAdjacencyMatrix[g].ToAdjacencyMatrix[g]])
Out[114]//TableForm= 3   1   1   0
                     1   2   1   1
                     1   1   2   1
                     0   1   1   1
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

No degree sequence can be graphic if all the degrees occur with multiplicity 1 [BC67]. Any sequence of positive integers summing up to an even number can be realized by a multigraph with self-loops [Hak62].

```
In[115]:= GraphicQ[{7,6,5,4,3,2,1}]
Out[115]= False
```

The direct implementation of the inductive definition of graphic degree sequences gives a deterministic algorithm for realizing any such sequence. However, `GraphicQ` provides a means to construct semirandom graphs of a particular degree sequence. We attempt to connect the vertex of degree Δ to a random set of Δ other vertices. If, after deleting 1 from the degrees of each of these vertices, the remaining sequence is graphic, then there exists a way to finish off the construction of the graph appropriately. If not, we keep trying other random vertex sets until it does.

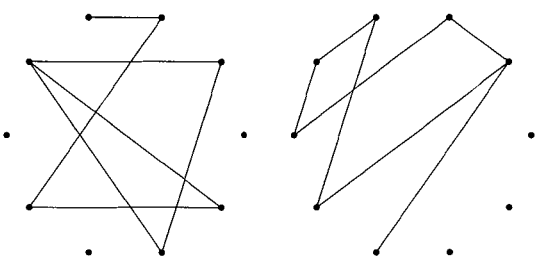
This algorithm does not pick a graph that realizes the given degree sequence, uniformly at random. Generating a graph that realizes a given degree sequence, uniformly at random, is a fairly hard problem [Luc92, MR98, MR95] that has drawn some attention lately due to applications in modeling the Web as a graph [ACL00, ACL01]. Our algorithm produces some graphs more often than others, but the function is still useful to produce test graphs satisfying certain degree constraints.

This is a graphic sequence, since it is the degree sequence of a graph.

```
In[116]:= d = DegreeSequence[g = RandomGraph[10, .25]]
Out[116]= {3, 2, 2, 2, 2, 2, 1, 0, 0, 0}
```

We apply `RealizeDegreeSequence` to produce graphs that realize this degree sequence. Since the function is randomized, its output may differ for different runs. If treated as labeled graphs, these two graphs are likely to be distinct. It is also possible that these are distinct unlabeled graphs.

```
In[117]:= ShowGraphArray[{g, RealizeDegreeSequence[d]}];
```



Most of these entries should be false, indicating that the generated graphs are nonisomorphic even if they have the same degree sequence.

```
In[118]:= Map[IsomorphicQ[g, #]&, Table[RealizeDegreeSequence[d], {10}]]
Out[118]= {False, True, False, False, False, False, False,
    False, False, False}
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Here RealizeDegreeSequence is called with a second argument that is an integer seed for the random number generator used in the function. Specifying a seed has the effect of making the function deterministic, so all five graphs generated here are identical.

```
In[119]:= Map[IdenticalQ[#, RealizeDegreeSequence[d,4]] &,
                Table[RealizeDegreeSequence[d,4], {5}]]
Out[119]= {True, True, True, True, True}
```

The construction is powerful enough to produce disconnected graphs when there is no other way to realize the degree sequence.

```
In[120]:= ConnectedComponents[ RealizeDegreeSequence[{2,2,2,1,1,1,1}] ]
Out[120]= {{1, 2, 3, 5, 7}, {4, 6}}
```

An important set of degree sequences are defined by *regular graphs*. A graph is regular if all vertices are of equal degree. An algorithm for constructing regular random graphs appears in [Wor84], although here we construct semirandom regular graphs as a special case of RealizeDegreeSequence.

```
RegularQ[g_Graph] := Apply[ Equal, Degrees[g] ]
RegularGraph[k_Integer, n_Integer] := RealizeDegreeSequence[Table[k,{n}]]
```

Constructing and Testing Regular Graphs

Here we construct and test a 4-regular graph on eight vertices. All cycles and complete graphs are regular.

```
In[121]:= RegularQ[ RegularGraph[4,8]]
Out[121]= True
```

Complete bipartite graphs are regular if and only if the two stages contain equal numbers of vertices.

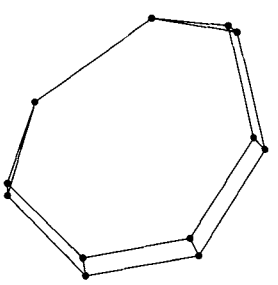
```
In[122]:= RegularQ[CompleteGraph[10,9]]
Out[122]= False
```

The join of two regular graphs is not necessarily regular.

```
In[123]:= DegreeSequence[GraphJoin[Cycle[4], CompleteGraph[2]]]
Out[123]= {5, 5, 4, 4, 4, 4}
```

Constructing regular graphs is a special case of realizing arbitrary degree sequences. Here we build a 3-regular graph of order 12. Such graphs are the adjacency graphs of convex polyhedra.

```
In[124]:= ShowGraph[g = RegularGraph[3, 12]; Nest[SpringEmbedding, g, 5]];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

6.5 Relations and Functional Graphs

The reason graphs are so important for modeling structures is that they are natural representations of binary relations. Here we provide tools for constructing graphs out of relationships between objects.

■ 6.5.1 Graphs from Relations

As defined in Section 3.3.3, a binary relation R on a set of objects S is a subset of the Cartesian product $S \times S$. Thus each element in R is an ordered pair of elements from S, and therefore R can be represented by a directed graph. `MakeGraph` takes a set of objects and a binary relation defined on the objects, and constructs a graph whose edges are defined by the binary relation. `MakeGraph` has quietly been used to construct many *Combinatorica* graphs, including butterfly, shuffle-exchange, and De Bruijn graphs.

The binary relation "u is not equal to v" defines a complete graph on the given set of objects. The relation is symmetric, and so the direction on edges is ignored by using the `Type` option.

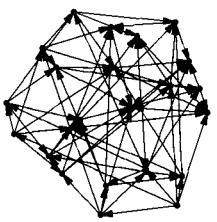
```
In[125]:= CompleteQ[ MakeGraph[Range[5], (#1!=#2)&, Type->Undirected]]
Out[125]= True
```

The *odd graphs* O_k [Big74] have vertices corresponding to the $k-1$ subsets of $\{1,\ldots,2k-1\}$ where two vertices are connected by an edge if and only if the associated subsets are disjoint. O_2 gives K_3, and here we prove that O_3 is the Petersen graph.

```
In[126]:= Isomorphism[PetersenGraph, MakeGraph[KSubsets[Range[5],2],
          (SameQ[Intersection[#1,#2], {}])&, Type -> Undirected]]
Out[126]= {1, 2, 8, 10, 7, 9, 6, 4, 5, 3}
```

Permutations differing by one transposition can be ordered by the number of inversions. The *inversion poset* [SW86] relates permutations p and q if there is a sequence of transpositions from p to q such that each transposition increases the number of inversions. Inversion posets and other partial orders are discussed in detail in Section 8.5. We can begin construction of this graph by linking all permutations that differ by one transposition in the direction of increasing disorder.

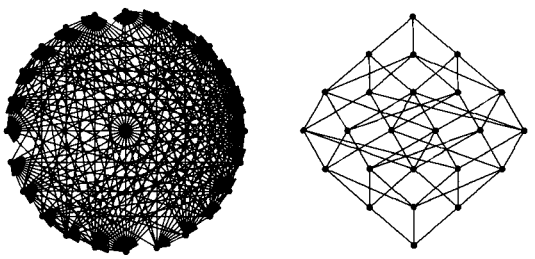
```
In[127]:= ShowGraph[SpringEmbedding[g=MakeGraph[ Permutations[{1,2,3,4}],
          (Count[#1-#2,0] == 2 && (Inversions[#1]>Inversions[#2]))&]]];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Taking the *transitive closure* of the previous graph adds an edge between any two vertices connected by a path, thus completing the construction of the inversion poset. The *Hasse diagram* of a poset eliminates those edges implied by other edges in the graph and ranks the vertices according to their position in the defined order. Here, the lowest ranked vertex corresponds to the identity permutation and the highest ranked vertex to its reverse. Each path from bottom to top through this lattice defines a sequence of transpositions that increase the number of inversions.

```
In[128]:= ShowGraphArray[{h = TransitiveClosure[g], HasseDiagram[h]}];
```

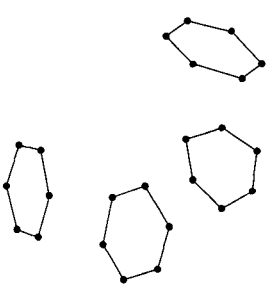


■ 6.5.2 Functional Graphs

A *functional graph* is a directed graph defined by a set of functions. More precisely, given a set S of objects and a set F of functions $f : S \to S$, the graph has vertex set S and for each $v \in S$ and each function $f \in F$ the graph contains a directed edge from v to $f(v)$.

Here, the vertices are 4-permutations with edges joining permutations π and π' if π' can be obtained by swapping either the first two elements or the second and third elements. `SpringEmbedding` reveals that the graph contains four disjoint 6-cycles, corresponding to the six 4-permutations that contain a given last element.

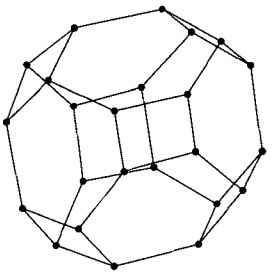
```
In[129]:= ShowGraph[SpringEmbedding[FunctionalGraph[
          {Permute[#, {2, 1, 3, 4}]&, Permute[#, {1, 3, 2, 4}] &},
          Permutations[4], Type -> Undirected]]];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

Adding one more function to the list of functions changes the structure of the graph dramatically. Now we get the familiar adjacent transposition graph. `MakeGraph` was used to construct this graph in Section 2.1.4.

```
In[130]:= ShowGraph[SpringEmbedding[FunctionalGraph[
          {Permute[#, {2, 1, 3, 4}] &, Permute[#, {1, 3, 2, 4}] &,
          Permute[#, {1, 2, 4, 3}] &}, Permutations[4],
          Type -> Undirected]]];
```



`FunctionalGraph` provides a convenient way of constructing *Cayley graphs*. A Cayley graph is defined by a group G and a set of generators S for G. The vertices of the graph are elements of G, and there is an edge from group element g to group element h if there is a generator s in S such that $g \times s = h$.

Here is a set of generators for the symmetric group of order 4. Each generator is obtained by swapping an element with the first element.

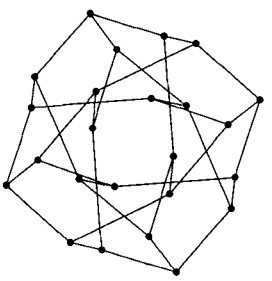
```
In[131]:= s=Table[p=Range[4]; {p[[1]],p[[i]]}={p[[i]],p[[1]]};
          p, {i,2,4}]
Out[131]= {{2, 1, 3, 4}, {3, 2, 1, 4}, {4, 2, 3, 1}}
```

These generators are converted into "functional forms" that can be used as the set of functions in `FunctionalGraph`.

```
In[132]:= l = Map[Function[x, Permute[x, #]] &, s]
Out[132]= {Function[x, Permute[x, {2, 1, 3, 4}]],
     Function[x, Permute[x, {3, 2, 1, 4}]],
     Function[x, Permute[x, {4, 2, 3, 1}]]}
```

Here is the Cayley graph defined by the above set of generators. It is a cubic graph because there are three generators.

```
In[133]:= g = FunctionalGraph[l, Permutations[4], Type -> Undirected];
          ShowGraph[Nest[SpringEmbedding, g, 10]];
```



https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

The diameter of this graph is the number of operations needed to sort a permutation, assuming each operation swaps an element with the first element.

```
In[135]:= Map[(UnrankPermutation[#-1,4])&, ShortestPath[g,
          RankPermutation[{1,2,3,4}]+1,
                RankPermutation[{4,3,2,1}]+1]]
Out[135]= {{1, 2, 3, 4}, {2, 1, 3, 4}, {3, 1, 2, 4},
     {1, 3, 2, 4}, {4, 3, 2, 1}}
```

Both `MakeGraph` and `FunctionalGraph` can be used to construct the same graph. Sometimes one is more convenient than the other.

```
In[136]:= n = 300;
          Timing[h = MakeGraph[Range[0, n - 1],
            (Mod[#1 + 1, n] == #2) || (Mod[#1 + 2, n] == #2) &]]
Out[137]= {11.5 Second, -Graph:<600, 300, Directed>-}
```

`FunctionalGraph` is faster for sparse graphs, since `MakeGraph` fills the adjacency matrix of the graph and thus takes $O(n^2)$ time independent of the size of the graph.

```
In[138]:= Timing[g = FunctionalGraph[{Mod[# + 1, n] &, Mod[# + 2, n] &},
          Range[0, n - 1]]]
Out[138]= {0.31 Second, -Graph:<600, 300, Directed>-}
```

The two graphs constructed above are indeed identical.

```
In[139]:= IdenticalQ[g, h]
Out[139]= True
```

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

6.6 Exercises

6.6.1 Thought Exercises

1. Prove that the graph product operation is commutative, that is, $G \times H$ is isomorphic to $H \times G$.
2. Is the graph product of two Hamiltonian graphs Hamiltonian? Is the graph product of two Eulerian graphs Eulerian? Prove your results.
3. Prove that the line graph of an Eulerian graph is both Eulerian and Hamiltonian, while the line graph of a Hamiltonian graph is always Hamiltonian.
4. Under what conditions is `CirculantGraph[m*n,{m,n}]` isomorphic to the graph product of cycles $C_m \times C_n$? Prove your results.
5. Prove that the contraction schemes described on butterfly graphs result in hypercubes.
6. Show that any r-bit binary string u can be converted to any other r-bit binary string v by $r - 1$ shuffles and at most r exchanges.
7. Show that the Prüfer codes of paths are exactly the sequences of $n - 2$ distinct integers.
8. What is the probability that a random labeled tree is a star?
9. Prove that the diagonal of the square of the adjacency matrix of a simple graph gives the unsorted degree sequence of the graph.
10. Prove that no degree sequence can be graphic if all the degrees occur exactly once.
11. Prove that any sequence of positive integers summing up to an even number can be realized by a multigraph with self-loops.
12. Give degree sequences such that any pair of graphs with such a degree sequence is isomorphic.

6.6.2 Programming Exercises

1. Give more direct implementations of `GraphUnion` and `GraphIntersection`. How does their performance compare with the existing versions?
2. Write a function to embed a hypercube in a grid graph, meaning to assign vertices of a hypercube to all the vertices of a grid graph such that the grid graph is contained within an induced subgraph of these hypercube vertices. How many of the $n!$ embeddings work?
3. The *total graph* $T(G)$ of a graph G has a vertex for each edge and each vertex of G, and an edge in $T(G)$ for every edge–edge and vertex–edge adjacency in G [CM78]. Thus total graphs are a generalization of line graphs. Write a function to construct the total graph of a graph.

https://doi.org/10.1017/CBO9781139164849.007 Published online by Cambridge University Press

4. Write functions to construct the following graphs from `GraphIntersection`, `GraphJoin`, `GraphUnion`, and `GraphProduct`: complete graphs, complete bipartite graphs, cycles, stars, and wheels.
5. What special cases of complete k-partite graphs are circulant graphs? Implement them accordingly.
6. Implement the function `RandomBipartiteGraph` that produces a labeled bipartite graph chosen uniformly at random from the set of all labeled bipartite graphs.
7. An alternate nondeterministic `RealizeDegreeSequence` can be based on the result [Egg75, FHM65] that every realization of a given degree sequence can be constructed from any other using a finite number of *edge-interchange* operations, where an edge interchange takes a graph with edges (x, y) and (w, z) and replaces them with edges (x, w) and (y, z). Implement a nondeterministic `RealizeDegreeSequence` function based on a sequence of edge-interchange operations.
8. A *degree set* for a graph G is the set of integers that make up the degree sequence. Any set of positive integers is the degree set for some graph. Design and implement an algorithm for constructing a graph that realizes an arbitrary degree set.

■ 6.6.3 Experimental Exercises

1. Experiment to determine the expected number of edges, vertices, and connected components remaining after deleting a random subset of the vertices of an $n \times m$ grid graph. How does the ratio of n to m effect the results?
2. Experiment with `RealizeDegreeSequence` to determine how many random subsets must be selected, in connecting the ith vertex of an order n graph, before finding one that leaves a graphic degree sequence.
3. Assign the vertices of a cycle C_n to a convex set of n points. A *swap* operation exchanges the points associated with two vertices of the graph. How many swaps are required to disentangle the graph and leave a planar embedding of the cycle?
4. What is the probability that a random unlabeled tree has a center of one vertex versus two vertices? For labeled trees, asymptotically half the trees are central and half bicentral [Sze83].
5. Consider a random graph with n vertices and edge probability p. Keep n fixed and increase p. This results in the graph becoming more dense. Experiment with `RandomGraph` to determine if there is a threshold probability $p(n)$ at which point isolated vertices disappear. Also, experiment to determine if there is a threshold probability at which the graph becomes connected.