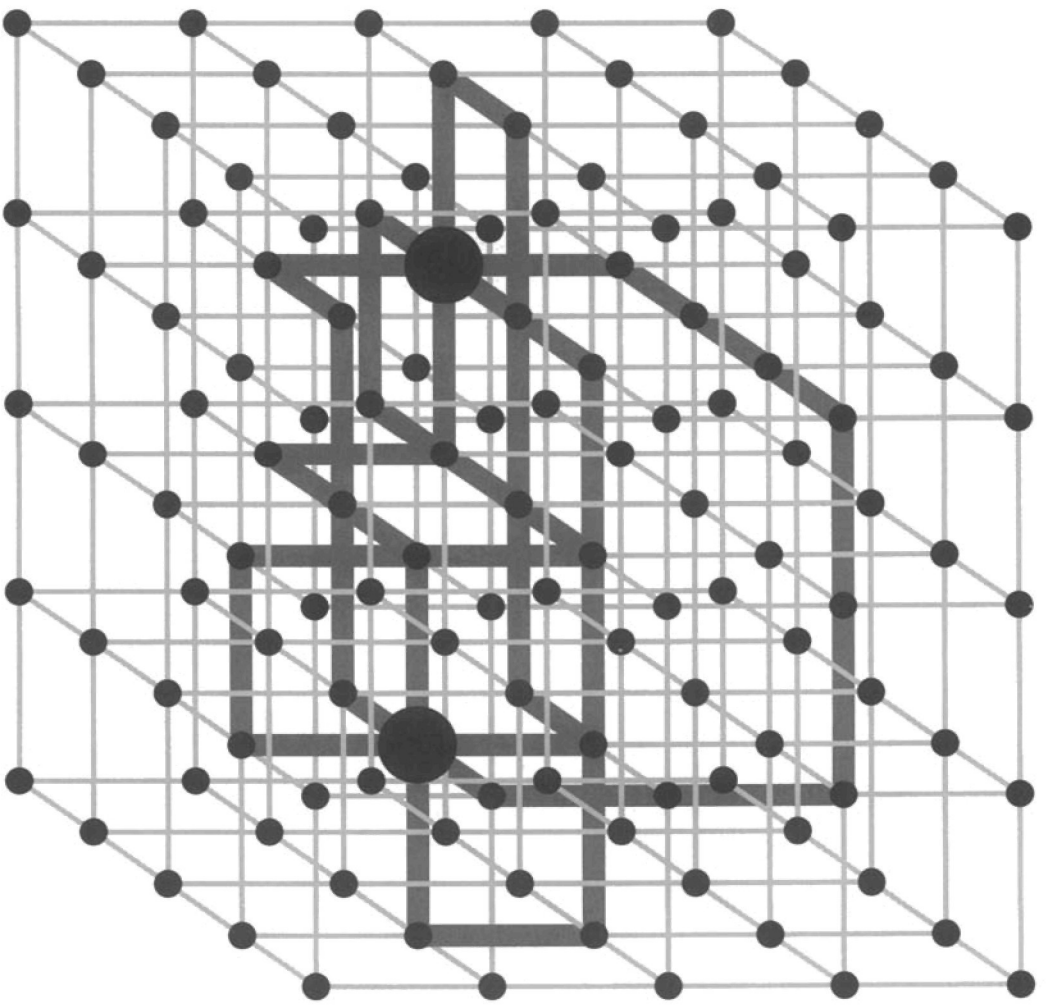
8. Algorithmic Graph Theory



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

Algorithmic graph theory is one of the best examples of mathematics inspired and nurtured by computer science. As we have seen, there is a natural division between problems that can be solved efficiently and problems that are hard. Most of the problems discussed in this chapter have efficient algorithms, and all are of practical importance.

Several fundamental graph algorithms are fairly complicated, and thus the implementations in the chapter are often longer and uglier than in the previous chapters. That is one reason for hiding them in the back of the book. This chapter is also where we pay most dearly for not having access to a traditional model of computation, since many of these algorithms make frequent modifications to data structures. Still, through clever programming we can compute interesting invariants on reasonably large graphs.

About the illustration overleaf:

One of the most important problems in algorithmic graph theory is computing the maximum amount of flow that can be sent from a source vertex to a target vertex, given capacity constraints on the edges. Many problems can be modeled as flow problems. Here we illustrate the computation of edge-disjoint paths between pairs of vertices using a solution to the maximum flow problem. There are six edge-disjoint paths between vertex 32 and vertex 93, as shown in this $5 \times 5 \times 5$ grid graph.

```
ShowGraph[Highlight[g = GridGraph[5, 5, 5], {{32, 93}, First[Transpose[
NetworkFlow[g, 32, 93, Edge]]]}], EdgeColor -> Gray];
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.1 Shortest Paths

The classic example of a real-life graph is a network of roads connecting cities. The most important algorithmic problem on such road networks is finding the shortest path between pairs of vertices.

In unweighted graphs, the length of a path is simply the number of edges in it. The shortest paths in such graphs can be found by using breadth-first search, as discussed in Section 7.1.1. Things get more complicated with weighted graphs, since a shortest path between two vertices might not be the one with the fewest edges.

There are several closely related variants of the shortest path problem. The *s-t shortest-path problem* seeks a shortest path from a source vertex s to a given sink t. The *single-source shortest-path problem* aims to find the shortest paths from s to all other vertices in the graph. The *all-pairs shortest-path problem* seeks the shortest paths from every vertex to every other vertex.

No one knows how to find s-t shortest paths without essentially solving the single-source shortest-path problem, so we focus on the latter. The all-pairs problem can, of course, be reduced to solving the single-source problem by using each vertex as the source, but the fastest algorithms work differently. We first focus on finding single-source shortest-paths before turning to the all-pairs problem.

■ 8.1.1 Single-Source Shortest Paths

There are two basic algorithms for finding single-source shortest paths. The simpler and more efficient *Dijkstra's algorithm* works correctly on graphs where no edges have negative weight. Negative-weight edges are rare in practice but not unheard of. The *Bellman-Ford algorithm* finds the shortest paths correctly in general-weighted graphs. Both are discussed below.

Dijkstra's Algorithm

Dijkstra's algorithm [Dij59], independently discovered by Whiting and Hillier [WH60], performs a "best-first search" on the graph G starting from source s. In the context of shortest paths, we will think of all graphs as directed and interpret each edge in an undirected graph as two directed edges going in opposite direction.

The algorithm maintains two distinct arrays to record the structure of the shortest paths in G:

- The distance array – For each vertex i, `dist[i]` maintains the length of the shortest known path from s to i. Clearly `dist`$[s] = 0$. For all vertices i distinct from s, `dist`$[i]$ is initialized to ∞.
- The parent array – For each vertex i, `parent[i]` maintains the predecessor of i on the shortest known path from s to i. The `parent` array is initialized by the algorithms to `parent`$[i] = i$ for all vertices i.

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

If `dist` contained the correct distances from s to every other vertex, then for any edge (i,j) with weight $w(i,j)$ we know that $\texttt{dist}[j] \le \texttt{dist}[i] + w(i,j)$. Any edge (i,j) for which this inequality holds is said to be *relaxed*. When Dijkstra's algorithm terminates, all edges are relaxed and `dist` has the correct shortest-path distances.

Dijkstra's algorithm goes through $n-1$ iterations. In each iteration it finds a shortest path from s to another new vertex. The efficiency of the algorithm depends on the following observation.

> Let S be the set of vertices to which the algorithm has already found shortest paths; that is, `dist` contains the correct values for every vertex $i \in S$. Furthermore, suppose that every edge outgoing from S is relaxed. Then, among all the vertices not in S, the vertex i with the smallest `dist` value has its `dist` value set correctly.

In each iteration, we scan the vertices not in S and pick a vertex i with the smallest `dist` value to include in S. Then we scan each edge (i,j) outgoing from i and if necessary relax it; that is, we check if $\texttt{dist}[j] > \texttt{dist}[i] + w(i,j)$ and, if so, reset $\texttt{dist}[j]$ to $\texttt{dist}[i] + w(i,j)$. This reset means we have found a shorter path to j, so $\texttt{parent}[j]$ is set to i.

When Dijkstra's algorithm ends, the edges between parents and their children define a *shortest-path spanning tree* of the graph, rooted at the source s. Our implementation returns both the distance and parent arrays.

```
Dijkstra[al_List, start_Integer] :=
      Module[{dist = Table[Infinity,{i, Length[al]}], parent = Table[i, {i, Length[al]}],
            untraversed = Range[Length[al]], m, v},
           dist[[start]] = 0;
           While[untraversed != {},
                m = Infinity;
                Scan[(If[dist[[#]]<=m, v=#;m=dist[[#]]])&, untraversed];
                untraversed = Complement[untraversed, {v}];
                n = Table[{al[[v, i, 1]],  m + al[[v, i, 2]]}, {i, Length[ al[[v]] ]}];
                Scan[If[dist[[ #[[1]] ]] > #[[2]], dist[[ #[[1]] ]] = #[[2]]; parent[[#[[1]]]] = v]&, n];
           ];
           {parent, dist}
      ]

Dijkstra[g_Graph, start_Integer] := Dijkstra[ToAdjacencyLists[g, EdgeWeight], start]

Dijkstra[g_Graph, start_List] :=
      Module[{al = ToAdjacencyLists[g, EdgeWeight]},
           Map[Dijkstra[ToAdjacencyLists[g, EdgeWeight], #]&, start]
      ]
```

Dijkstra's Shortest-Path Algorithm

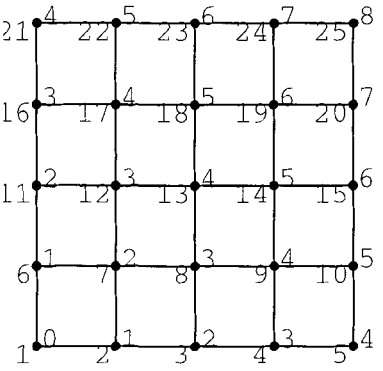
This loads the package.

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

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

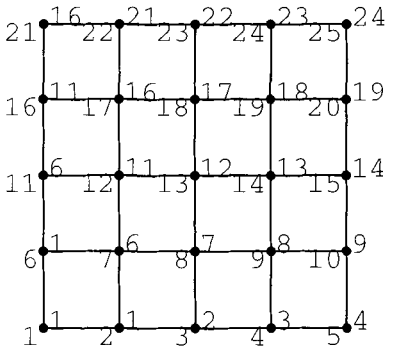
The *second* list returned by `Dijkstra` gives the distances from the source to all other vertices of the graph. The vertex numbers are displayed in the lower left corner, and the distances are displayed in the upper right corner of each vertex. In this unweighted graph, every edge has unit length, and the shortest paths minimize the number of edges. The "Manhattan" distance across an $n \times m$ grid graph is $n + m - 2$ and is computed here.

```
In[2]:= g = GridGraph[5, 5];
        ShowGraph[g, VertexLabel -> Dijkstra[g, 1][[2]],
        VertexNumber -> True, PlotRange -> 0.1,
        TextStyle->{FontSize->11}];
```



The first list returned by `Dijkstra` contains the parent relation in the shortest-path spanning tree. We use this information to label each vertex v with its parent on the shortest path from v to the source of the search. Observe that all parents are lower-numbered vertices, because the search originates from 1.

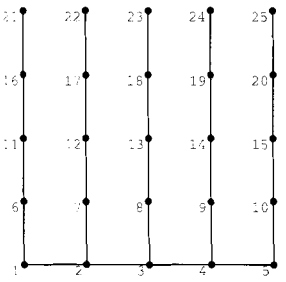
```
In[4]:= ShowGraph[g, VertexLabel -> Dijkstra[g, 1][[1]],
        VertexNumber -> True, PlotRange -> 0.15,
        TextStyle->{FontSize->11}];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

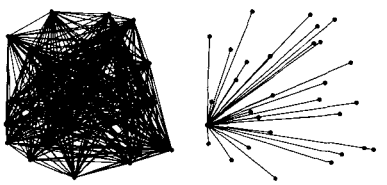
This parent relation is best illustrated by the *shortest-path spanning tree* defined by edges to parents. Since all edges are of equal weight, there are many different possible shortest-path spanning trees, depending on how the algorithm breaks ties.

```
In[5]:= ShowGraph[ShortestPathSpanningTree[g,1],
        VertexNumber->True, PlotRange->0.05];
```



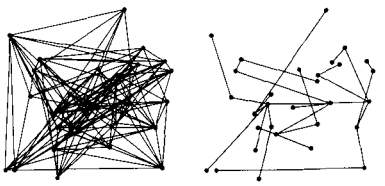
Here we run Dijkstra's algorithm on a 30-vertex graph whose edge weights equal the Euclidean distances between endpoints. The shortest path to every vertex is a direct hop of one edge from the root because the edge weights satisfy the triangle inequality.

```
In[6]:= g = SetEdgeWeights[ChangeVertices[RandomGraph[30,1],
        RandomVertices[30]], WeightingFunction -> Euclidean];
        ShowGraphArray[{g,ShortestPathSpanningTree[g,1]}];
```



Here we do the same experiment on a sparser random graph. Now there does not necessarily exist a direct route from the root to every vertex, so we see some branching and bending. Still, the tree reflects the geometry of the points. The root is identifiable as the vertex with high degree.

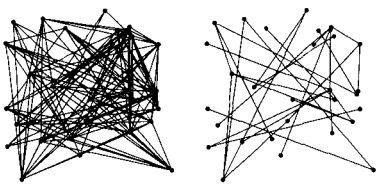
```
In[8]:= g = SetEdgeWeights[ChangeVertices[RandomGraph[30,0.3],
        RandomVertices[30]], WeightingFunction -> Euclidean];
        ShowGraphArray[{g,ShortestPathSpanningTree[g,1]}];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

With edge weights that are independent of the vertex position, the embedding of the spanning tree has many crossings. It is unclear which vertex is the root, because fewer neighbors link directly to it.

```
In[10]:= g = SetEdgeWeights[ChangeVertices[RandomGraph[30,0.3],
          RandomVertices[30]], WeightingFunction -> Random ];
          ShowGraphArray[{g,ShortestPathSpanningTree[g,1]}];
```

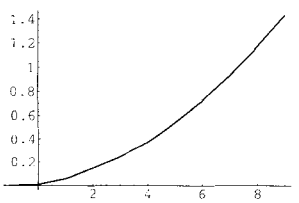


Dijkstra's algorithm works correctly on disconnected graphs as well. The infinities here are distances to vertices that cannot be reached from 1.

```
In[12]:= Dijkstra[GraphUnion[CompleteGraph[3], CompleteGraph[3]], 1]
Out[12]= {{1, 1, 1, 4, 5, 6},
    {0, 1, 1, Infinity, Infinity, Infinity}}
```

Dijkstra's algorithm runs in $\Theta(n^2)$ time on an n-vertex graph. This can be implemented in $O(m \log n)$ time by using binary heaps. By using the more sophisticated Fibonacci heap, it can be made to run in $O(n \log n + m)$ time on the RAM model of computation [FT87].

```
In[13]:= gt = Table[SetEdgeWeights[GridGraph[5, 5 i + 1] ], {i,0,10}];
          ListPlot[ Table[ {i-1, Timing[Dijkstra[gt[[i]], 1];][[1, 1]]},
          {i,0,10}], PlotJoined->True];
```



Bellman-Ford Algorithm

The Bellman-Ford algorithm correctly computes shortest paths even in the presence of negative-weight edges, provided the graph contains no negative cycles. A *negative cycle* is a directed cycle, the sum of whose edge weights is negative. This is clear trouble for any shortest-path algorithm, because we can make any path arbitrarily cheap by repeatedly cycling through this negative cost cycle.

Like Dijkstra's algorithm, the Bellman-Ford algorithm makes $(n-1)$ passes over the vertices, and on each pass it relaxes every edge that is not currently relaxed. However, unlike Dijkstra, Bellman-Ford scans all edges m in every pass to overcome negative edges.

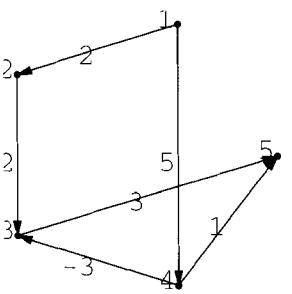
https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

To show the correctness of this algorithm, observe that after pass i the first $(i+1)$ vertices in any path from s have their `dist` values set correctly. If there are no negative cycles in the graph, $n-1$ is the maximum number of edges possible in any shortest path, so therefore $(n-1)$ passes suffice.

We implement an improvement of the Bellman-Ford algorithm due to Yen that uses $\lceil n/2 \rceil$ passes instead of $(n-1)$. This improvement depends on relaxing the edges in the graph in a certain order on each pass. Specifically, partition the set of edges E into "forward edges" $E_f = \{(i,j) \in E \mid i < j\}$ and "backward edges" $E_b = \{(i,j) \in E \mid i > j\}$. In each pass, first visit the vertices in the order $1, 2, \ldots, n$, relaxing the edges of E_f that leave each vertex. Then visit the vertices in the order $n, n-1, \ldots, 1$, relaxing the edges of E_b that leave each vertex. If there are no negative cycles, then $\lceil n/2 \rceil$ passes suffice to get all the `dist` values correct. This is implemented in the *Combinatorica* function `BellmanFord`.

This 5-vertex directed graph has only one negative-weight edge, but that is enough to cause trouble for Dijkstra's algorithm. It does not recognize that vertex 3 is more cheaply reached from vertex 1 via vertex 4 than via vertex 2.

```
In[15]:= g = FromOrderedPairs[{{1,2},{2,3},{1,4},{4,3},{4,5},{3,5}}];
         h = SetEdgeWeights[g, {2, 2, 5, -3, 1, 3}];
         ShowGraph[SetEdgeLabels[h, GetEdgeWeights[h]],
         VertexNumber -> True, VertexNumberPosition -> UpperLeft,
         TextStyle -> {FontSize -> 11}];
```



Dijkstra's algorithm tells us that going to vertex 5 via vertex 4 is best and costs us 6 units ...

```
In[18]:= TableForm[ Dijkstra[h, 1] ]
Out[18]//TableForm= 1   1   4   1   4
                    0   2   2   5   6
```

...while `BellmanFord` tells us that it only costs 5 units via vertex 3. Who do you believe?

```
In[19]:= TableForm[ BellmanFord[h, 1] ]
Out[19]//TableForm= 1   1   4   1   3
                    0   2   2   5   5
```

Adding the edge (5,2) with weight -7 creates a negative-weight cycle (2,3,5,2), so the shortest paths are no longer well-defined. Bellman-Ford returns an answer, but it is easy to verify that the answer is bogus since edge (2,3) is not relaxed.

```
In[20]:= h = AddEdges[h, {{5, 2}, EdgeWeight -> -7}];
         {par, dist} = BellmanFord[h, 1]
Out[21]= {{1, 5, 2, 1, 3}, {0, -5, -1, 5, 2}}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

Bellman-Ford can be used to detect the presence of negative cycles. Run the algorithm for $n - 1$ iterations and then test if there exists an edge that is not yet relaxed.

```
In[22]:= w = GetEdgeWeights[h]; e = Edges[h];
         Table[dist[[e[[i, 1]]]] + w[[i]] <= dist[[ e[[i, 2]]]], {i,
         Length[e]}]
Out[23]= {False, True, True, False, False, True, True}
```

The Bellman-Ford algorithm runs in $\Theta(mn)$ time on n-vertex, m-edge graphs. Both Dijkstra and Bellman-Ford have the same asymptotic running time on sparse graphs. In our experiments, Bellman-Ford beats Dijkstra, unless...

```
In[24]:= g = SetEdgeWeights[GridGraph[20, 20]];
         {Timing[Dijkstra[g, 1];], Timing[BellmanFord[g, 1];]}
Out[25]= {{4.15 Second, Null}, {1.11 Second, Null}}
```

...the input graph is sufficiently large and dense.

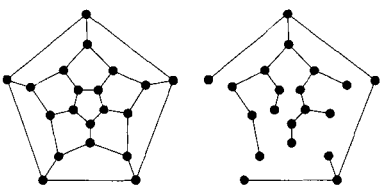
```
In[26]:= g = CompleteGraph[200];
         {Timing[Dijkstra[g, 1];], Timing[BellmanFord[g, 1];]}
Out[27]= {{10.83 Second, Null}, {15.11 Second, Null}}
```

Related Functions

Combinatorica provides a few "wrapper" functions that make it easier to use the shortest-path information. `ShortestPath` produces a sequence of vertices that define a shortest path from a given source to a given destination. `ShortestPathSpanningTree` produces a shortest-path spanning tree, containing the shortest paths from a source to all vertices in the graph. Both functions call `ChooseShortestPathAlgorithm` to determine whether to use Dijkstra's algorithm or Bellman-Ford. It makes its decision based on the presence of negative edge weights and the sparsity of the graph.

This shows the Euclidean shortest paths from vertex 1 to all other vertices in a dodecahedral graph. The dodecahedral graph is the "skeleton" of a dodecahedron, one of the five Platonic solids. The skeleton of any polyhedron is a planar graph. A *planar graph* can be drawn in the plane with no edge crossings. Shortest paths can be computed more quickly in planar graphs than in arbitrary graphs [Fre87, KRRS94].

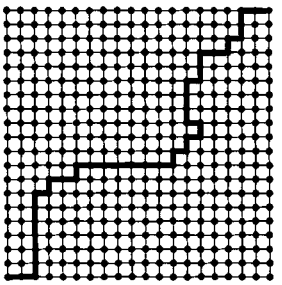
```
In[28]:= g = DodecahedralGraph;
         t = ShortestPathSpanningTree[SetEdgeWeights[g,
         WeightingFunction -> Euclidean], 1];
         ShowGraphArray[{g, t}, VertexStyle -> Disk[0.05]]
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

This shows a shortest path from the bottom left to the top right in a grid graph with random edge weights. Since the edge weights are random, there is no guarantee that the path will travel upward or to the right always. Can you determine how likely this is via experiment or analysis?

```
In[31]:= p = ShortestPath[g = SetEdgeWeights[GridGraph[20, 20]], 1, 400];
         ShowGraph[ Highlight[g, {Map[Sort, Partition[p, 2, 1]]}]];
```



■ 8.1.2 All-Pairs Shortest Paths

Shortest paths between every pair of vertices can be computed in $\Theta(n^3)$ time using Dijkstra's algorithm or in $\Theta(n^2 m)$ time using the Bellman-Ford algorithm repeatedly. We now present the *Floyd-Warshall algorithm* for computing all-pairs shortest paths. Asymptotically, it is as fast as Dijkstra's algorithm, however, it is much faster in practice because the algorithm is so short and simple.

```
AllPairsShortestPath[g_Graph] := {} /; (V[g] == 0)
AllPairsShortestPath[g_Graph] :=
      Module[{p = ToAdjacencyMatrix[g, EdgeWeight, Type->Simple], m},
            m = V[g]*Ceiling[Max[Cases[Flatten[p], _Real | _Integer]]]+1;
            Zap[DP1[p /. {0 -> 0.0, x_Integer -> 1.0 x, Infinity -> m*1.0}, m]] /. m -> Infinity
      ]

DP1 = Compile[{{p, _Real, 2}, {m, _Integer}},
      Module[{np = p, k, n = Length[p]},
            Do[np = Table[If[(np[[i, k]] == 1.0*m) || (np[[k, j]] == 1.0*m),
                        np[[i,j]], Min[np[[i,k]]+ np[[k,j]], np[[i,j]]]
                  ], {i,n},{j,n}
            ], {k, n}];
            np
      ]
     ]
```

The Floyd-Warshall Shortest Path Algorithm

The Floyd-Warshall algorithm [Flo62] is an application of dynamic programming. Single edges in the initial graph provide the shortest way to get between pairs of vertices with no intermediate stops. The algorithm successively computes the shortest paths between all pairs of vertices using the first k vertices as possible intermediaries, as k goes from 1 to n. The best way to get from s to t with $1,\ldots,k$

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

as possible intermediaries is the minimum of the best way using the first $k-1$ vertices and the best way known from s to t that goes through k. This gives an $\Theta(n^3)$-time algorithm that is essentially the same as the transitive closure algorithm to be discussed in Section 8.5.

We start by generating random integer weights in the range $[0, 10]$ for the edges of K_7 and use the Floyd-Warshall algorithm to compute the matrix of shortest-path distances between pairs of vertices.

```
In[33]:= g = SetEdgeWeights[ CompleteGraph[7],
          WeightingFunction -> RandomInteger, WeightRange -> {0, 10}];
          (s = AllPairsShortestPath[g]) // TableForm
Out[34]//TableForm= 0   3   4   4   3   5   2
                    3   0   6   6   6   5   5
                    4   6   0   2   6   6   6
                    4   6   2   0   4   6   6
                    3   6   6   4   0   2   5
                    5   5   6   6   2   0   7
                    2   5   6   6   5   7   0
```

Specifying the `Parent` tag as the second argument to `AllPairsShortestPath` produces parent information, in addition to shortest-path distances. The (i, j)th entry in the parent matrix contains the predecessor of j in a shortest path from i to j.

```
In[35]:= First[AllPairsShortestPath[g, Parent]] // TableForm
Out[35]//TableForm= 0   3   4   4   3   5   2
                    3   0   6   6   6   5   5
                    4   6   0   2   6   6   6
                    4   6   2   0   4   6   6
                    3   6   6   4   0   2   5
                    5   5   6   6   2   0   7
                    2   5   6   6   5   7   0
```

How much shorter does travel get if we take shortest paths rather than direct hops between pairs of vertices? The positive entries tell us that, for those pairs of vertices, the edge directly connecting them is not the shortest path.

```
In[36]:= (ToAdjacencyMatrix[g, EdgeWeight] - s) /.
          Infinity -> 0 // TableForm
Out[36]//TableForm= 0   0   0   0   0   4   0
                    0   0   0   0   0   0   1
                    0   0   0   0   0   0   3
                    0   0   0   0   0   4   2
                    0   0   0   0   0   0   2
                    4   0   0   4   0   0   1
                    0   1   3   2   2   1   0
```

Although Floyd-Warshall has the same worst-case time complexity as n calls to Dijkstra, it is much faster in practice.

```
In[37]:= g = RandomGraph[30,0.5];
          {Timing[ Table[ Dijkstra[g,i], {i,1,V[g]}];] [[1,1]],
          Timing[ AllPairsShortestPath[g];] [[1,1]]}
Out[38]= {4.08, 0.59}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

■ 8.1.3 Applications of All-Pairs Shortest Paths

Several graph invariants depend on the all-pairs shortest-path matrix. The *eccentricity* of a vertex v in a graph is the length of the longest shortest path from v to some other vertex. From the eccentricity come other graph invariants. The *radius* of a graph is the smallest eccentricity of any vertex, while the *center* is the set of vertices whose eccentricity is the radius. The *diameter* of a graph is the maximum eccentricity of any vertex.

In the following we present functions that compute these graph invariants. We try to improve the efficiency of `Eccentricity` by distinguishing between weighted and unweighted graphs. In the latter case, it is sufficient to call the breadth-first search function to compute distances.

```
Eccentricity[g_Graph, start_Integer, NoEdgeWeights] := Max[ BreadthFirstTraversal[g, start, Level] ]
Eccentricity[g_Graph, start_Integer] := Eccentricity[g, start, NoEdgeWeights] /; UnweightedQ[g]
Eccentricity[g_Graph, start_Integer] := Map[Max, Last[BellmanFord[g, start]]]
Eccentricity[g_Graph] := Table[Eccentricity[g, i, NoEdgeWeights], {i, V[g]}] /; UnweightedQ[g]
Eccentricity[g_Graph] := Map[ Max, AllPairsShortestPath[g] ]

Radius[g_Graph] := Min[ Eccentricity[g] ]
Diameter[g_Graph] := Max[ Eccentricity[g] ]
GraphCenter[g_Graph] :=
      Module[{eccentricity = Eccentricity[g]},
            Flatten[ Position[eccentricity, Min[eccentricity]] ]
      ]
```

Finding the Eccentricity, Diameter, Radius, and Center

In graph theory as in business, high eccentricity is not characteristic of a wheel.

```
In[39]:= Eccentricity[ Wheel[10] ]
Out[39]= {2, 2, 2, 2, 2, 2, 2, 2, 2, 1}
```

A wheel has diameter 2 and radius 1, because a path through the center connects any pair of vertices.

```
In[40]:= {Diameter[Wheel[10]], Radius[Wheel[10]]}
Out[40]= {2, 1}
```

The diameter and the radius of disconnected graphs are infinite.

```
In[41]:= {Diameter[g = GraphUnion[Star[4], Star[4]]], Radius[g]}
Out[41]= {Infinity, Infinity}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

Here we calculate the diameter of random graphs. The graphs corresponding to row i have an edge probability of $1/i$. So the graphs in the first row are completely connected, and as we go down the rows the graphs become more and more sparse. The corresponding increase in diameter is evident. To make the table more compact, infinities have been replaced by -1's.

```
In[42]:= (Table[Diameter[RandomGraph[20, 1/i]], {i, 10}, {5}]
            /. Infinity -> -1) // TableForm
Out[42]//TableForm= 1    1    1    1    1
                    2    3    3    2    2
                    3    3    3    3    3
                    4    4    4    4    3
                    4    4    5    -1   4
                    5    -1   5    4    4
                    4    -1   -1   -1   -1
                    5    -1   -1   -1   -1
                    -1   -1   -1   -1   -1
                    -1   -1   -1   -1   -1
```

Because the distance function satisfies the triangle inequality, the diameter of a graph is at most twice its radius.

```
In[43]:= g = RandomGraph[30,0.2]; {Diameter[g], Radius[g]}
Out[43]= {4, 3}
```

The cycle does not have a unique center like the wheel does.

```
In[44]:= GraphCenter[ Cycle[10] ]
Out[44]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

The center of any tree always consists of either one vertex or two adjacent vertices. Note: The two adjacent vertices in a tree do not necessarily have adjacent labels assigned to them.

```
In[45]:= Table[ GraphCenter[RandomTree[30]], {5}]
Out[45]= {{29}, {16}, {19}, {7, 22}, {14}}
```

■ 8.1.4 Number of Paths

The kth *power* of a graph G is a graph with the same set of vertices as G and an edge between two vertices if and only if there is a path of length at most k between them. Since a path of length 2 between u and v exists for every vertex v' such that $\{u, v'\}$ and $\{v', v\}$ are edges in G, the square of the adjacency matrix of G counts the number of such paths. By induction, the (u, v)th element of the kth power of the adjacency matrix of G gives the number of paths of length k between vertices u and v. Therefore, summing up the first k powers of the adjacency matrix counts all paths of length up to k.

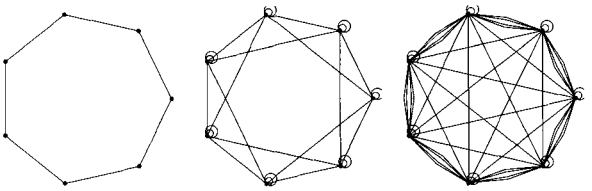
```
GraphPower[g_Graph,1] := g
GraphPower[g_Graph, k_Integer] :=
      Module[{prod = power = p = ToAdjacencyMatrix[g]},
            FromAdjacencyMatrix[Do[prod = prod.p; power=power+prod, {k-1}]; power, Vertices[g, All]]
      ]
```

Computing the Power of a Graph

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

The diameter of a cycle on seven vertices is 3, so the cube of such a graph is complete. There are four distinct but nonsimple paths of length at most 3 between adjacent vertices in a cycle. These show up as edges connecting adjacent vertices in the cube of the cycle. The two self-loops at each vertex are the two distinct ways of going from a vertex to itself in three or fewer hops.

```
In[46]:= ShowGraphArray[ Table[GraphPower[Cycle[7], i], {i,1,3}] ];
```



The number of shortest paths between opposite corners of an $m \times n$ grid graph is $\binom{n+m-2}{m-1}$, as shown in the last entry of the first row. Which entry in the matrix is largest, and why?

```
In[47]:= ToAdjacencyMatrix[GraphPower[GridGraph[3, 3],4]] // TableForm
Out[47]//TableForm=
```

12	6	9	6	18	3	9	3	6
6	21	6	18	9	18	3	15	3
9	6	12	3	18	6	6	3	9
6	18	3	21	9	15	6	18	3
18	9	18	9	36	9	18	9	18
3	18	6	15	9	21	3	18	6
9	3	6	6	18	3	12	6	9
3	15	3	18	9	18	6	21	6
6	3	9	3	18	6	9	6	12

Fleischner [Fle74] showed that the square of any biconnected graph is Hamiltonian.

```
In[48]:= HamiltonianCycle[GraphPower[CompleteGraph[10,2], 2]]
Out[48]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1}
```

The cube of any connected graph is also Hamiltonian.

```
In[49]:= HamiltonianCycle[GraphPower[RandomTree[15], 3]]
Out[49]= {1, 4, 6, 7, 2, 3, 10, 5, 9, 11, 14, 12, 13, 15,
    8, 1}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.2 Minimum Spanning Trees

A *minimum spanning tree* (MST) [GH85] of an edge-weighted graph is a set of $n-1$ edges of minimum total weight that form a spanning tree of the graph. The two classical algorithms for finding such a tree are *Prim's algorithm* [Pri57], which greedily adds edges of minimum weight that extend the existing tree and do not create cycles, and *Kruskal's algorithm* [Kru56], which greedily adds edges that connect components. We implement Kruskal's algorithm.

Students sometimes cast a jaded eye at the minimum spanning tree problem because a simple greedy approach suffices for finding the optimal solution. However, it is quite remarkable that a minimum spanning tree can be computed in polynomial time. Many simple variants of the problem, such as finding a spanning tree with maximum degree k or a spanning tree minimizing the total length between all pairs of vertices, are NP-complete [GJ79]. The minimum spanning tree problem can be formulated as a *matroid* [PS82], a system of independent sets whose largest weighted independent set can be found by using the greedy algorithm.

8.2.1 Union-Find

Kruskal's algorithm repeatedly picks the lowest-weight edge and tests whether it bridges two connected components in the current partial tree. If so, then the two components are merged and the edge is added to the partial tree. Thus components can be maintained as a collection of disjoint sets supporting two operations: `FindSet`, which returns the name of the set containing a given element, and `UnionSet`, which, given the names of two sets, merges them into one. When a new candidate edge (i, j) arrives, we find the names of the sets containing i and j by making calls to `FindSet`. We then check if the two sets are identical – if not, we merge them into one by calling `UnionSet`.

Our data structure maintains each set in the collection as a tree such that the name of each set is its root. `UnionSet` merges two sets by assigning the bigger tree to be the root of the shorter one, thus giving the sets the same name while minimizing the height of the tree for efficiency. Because the smaller tree becomes a child of the bigger one on each union, the trees are balanced and `FindSet` can be shown to take $O(\log n)$ time per operation. The `UnionSet` operation makes two calls to `FindSet` and then spends $O(1)$ extra time to connect the two sets together, and so this operation also takes $O(\log n)$ time. Adding *path compression* [Tar75] would make the data structure even more efficient.

```
InitializeUnionFind[n_Integer] := Module[{i}, Table[{i,1},{i,n}] ]

FindSet[n_Integer,s_List] :=
      Block[{$RecursionLimit = Infinity}, If [n == s[[n,1]], n, FindSet[s[[n,1]],s]]]

UnionSet[a_Integer,b_Integer,s_List] :=
      Module[{sa=FindSet[a,s], sb=FindSet[b,s], set=s},
            If[ set[[sa,2]] < set[[sb,2]], {sa,sb} = {sb,sa} ];
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

```
            set[[sa]] = {sa, Max[ set[[sa,2]], set[[sb,2]]+1 ]};
            set[[sb]] = {sa, set[[sb,2]]};
            set
      ]
```

The Union-Find Data Structure

Element i is represented by an ordered pair in the ith position of a list. The first element is a parent pointer, while the second is the height of the subtree rooted at that node. This example contains three sets: a singleton containing 1; a singleton containing 2; and a set containing 3, 4, and 5.

```
In[50]:= UnionSet[3,4, UnionSet[4,5, InitializeUnionFind[5] ] ]
Out[50]= {{1, 1}, {2, 1}, {4, 1}, {4, 2}, {4, 1}}
```

The name of the root is not particularly significant. Here the first set is called 1, the second is called 2, and the third is called 4.

```
In[51]:= Table[FindSet[i,%], {i, 5}]
Out[51]= {1, 2, 4, 4, 4}
```

■ 8.2.2 Kruskal's Algorithm

Kruskal's algorithm sorts the edges in order of increasing cost and then repeatedly adds edges that bridge components until the graph is fully connected. The union-find data structure maintains the connected components in the forest. That this algorithm gives a minimum spanning tree follows because in any cycle the most expensive edge is the last one considered, and thus it cannot appear in the minimum spanning tree. The code for `MinimumSpanningTree` is given below. By negating the weights for each edge, this implementation can also be used to find a maximum-weight spanning tree.

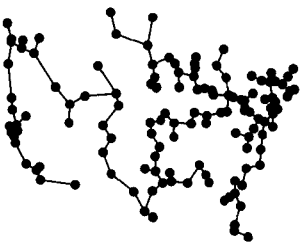
```
MinimumSpanningTree[e_List, g_Graph] :=
      Module[{ne=Sort[e, (#1[[2]] <= #2[[2]])&],
            s=InitializeUnionFind[V[g]]},
            ChangeEdges[g,
                  Select[Map[First, ne],
                        (If[FindSet[#[[1]],s]!=FindSet[#[[2]], s],
                            s=UnionSet[#[[1]],#[[2]], s]; True, False
                         ])&
                  ]
            ]
      ]
MinimumSpanningTree[g_Graph] := MinimumSpanningTree[ Edges[g, EdgeWeight], g ] /; UndirectedQ[g]

MaximumSpanningTree[g_Graph] := MinimumSpanningTree[Map[{First[#], -Last[#]}&, Edges[g, EdgeWeight]], g]
```

Finding a Minimum Spanning Tree

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

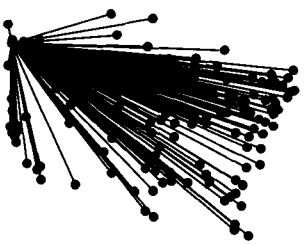
Minimum spanning trees are useful in encoding the gross structure of geometric graphs. Here we take the minimum spanning tree of 128 North American cities and can clearly recognize the outline of the United States.

```
In[52]:= usa = SetVertexLabels[<< "extraCode/usa.graph",{" "}]
         ShowGraph[mst = MinimumSpanningTree[usa] ];
```



The shortest-path spanning tree on such a complete graph is a star. This, of course, obscures the outline and the internal structure of the map.

```
In[54]:= ShowGraph[spt = ShortestPathSpanningTree[usa,55] ];
```



This reports the total costs of the minimum spanning tree and the shortest-path spanning tree. By definition, the cost of the shortest-path spanning tree should be at least as large. The difference here is very dramatic because the U.S. graph was complete.

```
In[55]:= {Apply[ Plus, GetEdgeWeights[usa, Edges[mst]]],
          Apply[Plus, GetEdgeWeights[usa, Edges[spt]]]}
Out[55]= {16598, 259641}
```

The maximum spanning tree must be no smaller than the shortest-path spanning tree, and should generally be heavier.

```
In[56]:= Apply[Plus, GetEdgeWeights[usa, Edges[MaximumSpanningTree[usa]]]]
Out[56]= 341365
```

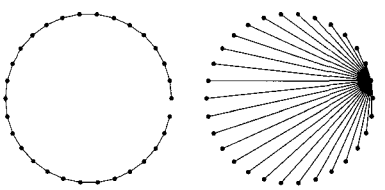
https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

The minimum spanning tree not only minimizes the sum of the costs of the edges, it also minimizes the cost of the maximum edge or the "bottleneck" edge. Thus the minimum spanning tree is also a *minimum bottleneck spanning tree.*

```
In[57]:= {Max[GetEdgeWeights[usa, Edges[mst]]], Max[GetEdgeWeights[usa,
          Edges[spt]]]}
Out[57]= {423, 3408}
```

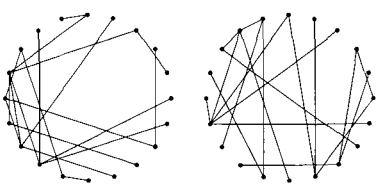
A *Euclidean minimum spanning tree* is the MST of a complete graph whose vertices are points in Euclidean space and whose edges have Euclidean distances as weights. A Euclidean MST for a set of points evenly distributed on the circumference of a circle is always a path along the circumference, while every shortest-path spanning tree is a star.

```
In[58]:= g = SetEdgeWeights[CompleteGraph[30], WeightingFunction ->
          Euclidean];
          ShowGraphArray[{MinimumSpanningTree[g], ShortestPathSpanningTree[g,
          1]} ];
```



One way to produce random spanning trees is to assign random weights to edges and run the minimum spanning tree algorithm on it. Here we show two runs of this on K_{20}.

```
In[60]:= g = CompleteGraph[20];
          ShowGraphArray[Map[MinimumSpanningTree[SetEdgeWeights[#]] &, {g,
          g}]];
```



In a large complete graph with random edge weights, it is unlikely that the maximum and minimum spanning trees will have any edges in common.

```
In[62]:= g = SetEdgeWeights[ CompleteGraph[20], WeightingFunction -> Random];
          M[GraphIntersection[ MinimumSpanningTree[g],
          MaximumSpanningTree[g]]]
Out[63]= 0
```

Prim's algorithm consists of $(n-1)$ iterations, each of which takes $O(n)$ time to select a smallest edge connecting a vertex in the connected component to one that is not. This total time complexity of $O(n^2)$ is optimal on dense graphs. The trick is maintaining a shortest edge to each outside vertex in an array, so, after adding the ith edge to the tree, only $n-i-1$ edges must be tested to update this array. More efficient algorithms rely on sophisticated data structures [FT87] or randomization [KKT95].

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

■ 8.2.3 Counting Spanning Trees

The number of spanning trees of a graph G was determined by Kirchhoff [Kir47] by using certain operations on the adjacency matrix of G.

The (i,j) *minor* of a matrix M is the determinant of M with the ith row and jth column deleted. The (i,j) *cofactor* of a matrix M is $(-1)^{i+j}$ times the (i,j) minor of M.

```
Cofactor[m_?MatrixQ, {i_Integer?Positive , j_Integer?Positive}] :=
      (-1)^(i+j) * Det[ Drop[ Transpose[ Drop[Transpose[m],{j,j}]], {i,i}]] /; (i <= Length[m]) &&
                                                                  (j <= Length[m[[1]]])
```

Computing a Cofactor of a Matrix

The number of spanning trees of a graph G is equal to any cofactor of the degree matrix of G minus the adjacency matrix of G, where the degree matrix of a graph is a diagonal matrix with the degree of v_i in the (i,i) position of the matrix. This *matrix-tree theorem* is due to Chaiken [Cha82].

```
NumberOfSpanningTrees[g_Graph] := 0 /; (V[g] == 0)
NumberOfSpanningTrees[g_Graph] := 1 /; (V[g] == 1)
NumberOfSpanningTrees[g_Graph] :=
      Module[{m = ToAdjacencyMatrix[g]},
            Cofactor[ DiagonalMatrix[Map[(Apply[Plus,#])&,m]] - m, {1,1}]
      ]
```

Counting Spanning Trees of a Graph

Any tree contains exactly one spanning tree.

```
In[64]:= NumberOfSpanningTrees[Star[20]]
Out[64]= 1
```

A cycle on n vertices contains exactly n spanning trees, since deleting any edge creates a tree.

```
In[65]:= NumberOfSpanningTrees[Cycle[20]]
Out[65]= 20
```

The number of spanning trees of a complete graph is n^{n-2}, as was shown by the Prüfer bijection between labeled trees and strings of integers discussed in Section 6.3.1.

```
In[66]:= NumberOfSpanningTrees[CompleteGraph[10]]
Out[66]= 100000000
```

The set of spanning trees of a graph can be ranked and unranked and thus can be generated systematically or randomly. See [CDN89] for $O(n^3)$ ranking and unranking algorithms.

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.3 Network Flow

We introduced shortest paths by discussing optimization on networks of roads. When we have a network of pipes and want to push sewage through the graph instead of cars, we have a *network flow* problem. More formally, a network flow problem consists of an edge-weighted graph G and *source* and *sink* vertices s and t. The weight of each edge signifies its capacity, the maximum amount of stuff that can be pumped through it. The *maximum-flow problem* seeks the maximum possible flow from s to t, satisfying the capacity constraint at each edge and satisfying the constraint that at each vertex other than s and t the net flow through the vertex should be 0. Network flow is important because many other problems can be easily reduced to it, including k-connectivity and bipartite matching.

Traditional network flow algorithms use the *augmenting path* idea of Ford and Fulkerson [FF62], which repeatedly finds a path of positive capacity from s to t and adds it to the flow. It can be shown that the flow through a network of rational capacities is optimal if and only if it contains no augmenting path. Since each augmentation adds to the flow, if all capacities are rational, we will eventually find the maximum. However, each augmenting path may add but a little to the total flow, and so the algorithm might take a long time to converge.

Edmonds and Karp [EK72] proved that always selecting a *shortest* geodesic augmenting path guarantees that $O(n^3)$ augmentations suffice for optimization. The Edmonds-Karp algorithm is fairly easy to implement, since a breadth-first search from the source can find a shortest path in linear time.

The key structure is the *residual flow graph*, denoted $R(G,f)$, where G is the input graph and f is the current flow through it. This directed, edge-weighted $R(G,f)$ has the same vertices as G and for each edge (i,j) in G with capacity $c(i,j)$ and flow $f(i,j)$, $R(G,f)$ may contain two edges:

(i) an edge (i,j) with weight $c(i,j) - f(i,j)$, if $c(i,j) - f(i,j) > 0$ and

(ii) an edge (j,i) with weight $f(i,j)$, if $f(i,j) > 0$.

Edge (i,j) in the residual graph indicates that a positive amount of flow can be pushed from i to j. The weight of the edge gives the exact amount that can be pushed. A path in the residual flow graph from s to t implies that more flow can be pushed from s to t and the minimum weight of an edge in this path equals the amount of extra flow that can be pushed.

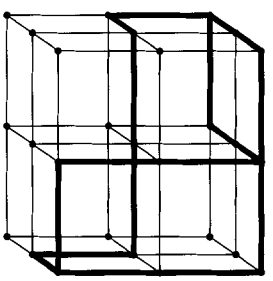
In the *Combinatorica* function `NetworkFlow`, we repeatedly start with a flow, construct the residual flow graph for it, perform a breadth-first search on this graph to find a path from s to t, and push the flow along that path, thereby ending up with a new and improved flow. The algorithm starts with a flow of 0 through every edge, and the algorithm terminates when the residual flow graph has no path from s to t.

Here we compute the flow across a cube with all edge capacities equal to 1. This flow is returned as an augmented adjacency list, where the flow appears along with each edge. For example, vertex 1 has neighbors 2, 3, and 5, and in the maximum flow a unit flow is pushed from 1 to each of its neighbors.

```
In[67]:= NetworkFlow[g = GridGraph[2, 2, 2], 1, 8, All] // ColumnForm

Out[67]= {{2, 1}, {3, 1}, {5, 1}}
         {{1, 0}, {4, 1}, {6, 0}}
         {{1, 0}, {4, 0}, {7, 1}}
         {{2, 0}, {3, 0}, {8, 1}}
         {{1, 0}, {6, 1}, {7, 0}}
         {{2, 0}, {5, 0}, {8, 1}}
         {{3, 0}, {5, 0}, {8, 1}}
         {{4, 0}, {6, 0}, {7, 0}}
```

The connection between network flows and edge connectivity is illustrated here. In any graph whose edge capacities are all 1, the maximum flow from s to t corresponds to the number of edge-disjoint paths from s to t. Specifically, the edges along which positive flow is pushed form paths from s to t. Here three units of flow can be pushed from vertex 1 to vertex 27 along the edge-disjoint paths shown.

```
In[68]:= ShowGraph[Highlight[g = GridGraph[3, 3, 3],
         {First[Transpose[NetworkFlow[g, 1, 27, Edge]]]}]];
```



Here we construct the directed edge-weighted graph that will be our running network flow example.

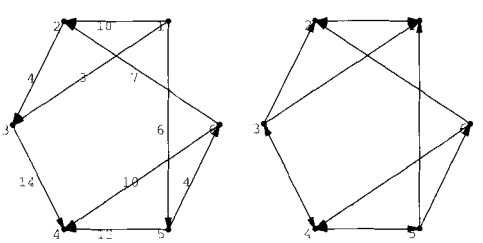
```
In[69]:= g = AddEdges[EmptyGraph[6, Type -> Directed],
                      {{1, 2}, {2, 3}, {1, 3}, {3, 4}, {1, 5},
                       {5, 6}, {5, 4}, {6, 4}, {6, 2}} ];
         h = SetEdgeWeights[g, {10, 4, 3, 14, 6,  4, 12, 10, 7}];
         h = SetEdgeLabels[h,GetEdgeWeights[h]];
```

Calling `NetworkFlow` tells us that 13 units of flow will be sent from vertex 1 to vertex 4.

```
In[72]:= NetworkFlow[h, 1, 4]

Out[72]= 13
```

These are the original graph (left) and residual flow graph (right) corresponding to the maximum flow from vertex 1 to vertex 4. There is no directed 1-to-4 path in the residual graph, because such a path means that more flow can be pushed from source to sink. Only vertices 1 and 2 remain reachable from the source, yielding a source–sink partition $S = \{1, 2\}$ and $T = \{3, 4, 5, 6\}$. All edges that go from S to T are *saturated*, meaning they carry the maximum possible flow. Similarly, edges from T to S in the residual graph carry no flow.

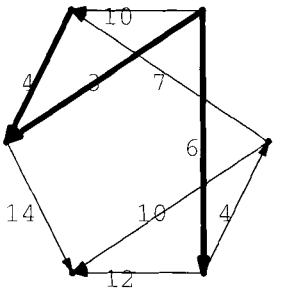
```
In[73]:= f = NetworkFlow[h, 1, 4, All];
         ShowGraphArray[{h,ResidualFlowGraph[h,f]}, VertexNumber->True];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

A *minimum s-t cut* in an edge-weighted graph is a set of edges with minimum total weight whose removal separates vertex s from vertex t. The *max-flow min-cut* theorem states that the weight of a minimum s-t cut equals the maximum flow that can be sent from s to t when interpreting the weights as capacities. The minimum s-t cut is defined by the source–sink partition above. Specifically, the edges from S to T give the minimum s-t cut.

```
In[75]:= cut = NetworkFlow[h, 1, 4, Cut];
         ShowGraph[Highlight[h, {cut}], TextStyle->{FontSize->10}];
```



Here the edge connectivity of the graph matches that of the s-t cut, but in general we need to minimize over all possible source–sink cuts.

```
In[77]:= EdgeConnectivity[ MakeUndirected[g] ]
Out[77]= 3
```

A fast algorithm for 0-1 network flow appears in [ET75]. This is useful for problems that use network flow as a subroutine. Such algorithms often require solving flow problems between every pair of vertices in the graph. For undirected graphs, $n-1$ applications of the maximum-flow algorithm suffice to determine all $\binom{n}{2}$ pairwise flows [GH61].

Active research continues in finding better network flow algorithms, with [Tar83] an excellent text and [AMO93] reflective of the state of the art.

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.4 Matching

A *matching* in a graph G is a set of edges of G no two of which have a vertex in common. Clearly, every matching consists of at most $n/2$ edges; matchings with exactly $n/2$ edges are called *perfect*. Not all graphs have perfect matchings, but every graph has a *maximum* or largest matching.

A perfect matching on a graph is a 1-regular subgraph of order n. In general, a *k-factor* of a graph is a k-regular subgraph of order n. k-factors are a generalization of perfect matchings, since perfect matchings are 1-factors. A graph is *k-factorable* if it is the union of disjoint k-factors.

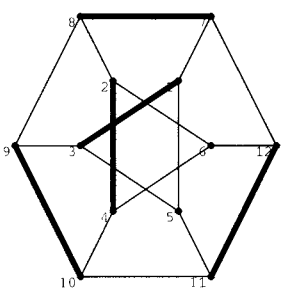
■ 8.4.1 Maximal Matching

The problem of finding matchings is naturally generalized to edge-weighted graphs, where the goal is to find a matching that maximizes the sum of the weights of the edges. An excellent reference on matching theory and algorithms is [LP86].

A *maximal matching* is one that cannot be enlarged by simply adding an edge. Maximal matchings are easy to compute: Repeatedly pick an edge disjoint from those already picked until this is no longer possible. Maximal matchings need not be maximum matchings. Indeed, a graph may have maximal matchings of varying sizes and it is hard to find a maximal matching of smallest size. Fortunately larger matchings are usually more useful.

Here is a maximal matching in a generalized Petersen graph. Clearly, the matching is not perfect. Now walk along the path $5,3,1,7,8,9,10,11,12,6$, changing each matched edge to an unmatched and each unmatched edge to a matched. In the process, four matched edges are replaced by five matched edges, giving a perfect matching.

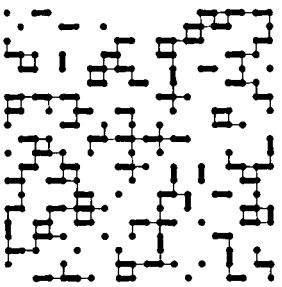
```
In[78]:= g = GeneralizedPetersenGraph[6, 4];
         ShowGraph[Highlight[g, {MaximalMatching[g]}],
         VertexNumber->True];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

A grid graph is bipartite and so is every induced subgraph. Here we construct a maximal matching in a random induced subgraph of a 20×20 grid graph. Is this a largest possible matching? Look for paths that start and end with unmatched vertices and contain edges that are alternately matched and unmatched. Flipping the status of edges in such a path increases the size of the matching by 1.

```
In[80]:= g = InduceSubgraph[GridGraph[20, 20], RandomSubset[400]];
         ShowGraph[Highlight[g, {mm=MaximalMatching[g]}]];
```

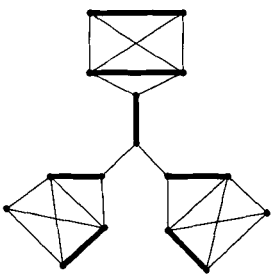


This shows the difference in sizes of a maximum matching and a maximal matching for the above graph.

```
In[82]:= {Length[BipartiteMatching[g]], Length[mm]}
Out[82]= {90, 79}
```

The maximal matching shown here is in fact maximum. Any larger matching would have to be perfect, but none exists. To see this, remove the central vertex. Three odd-sized components result, each of which needs the central vertex to match with a vertex in it. But the central vertex can match with only one vertex, leaving two components stranded.

```
In[83]:= ShowGraph[Highlight[g = NoPerfectMatchingGraph,
         {MaximalMatching[g]}]]
```



More generally, a graph $G = (V, E)$ has no perfect matching if there is a set $S \subseteq V$ whose removal results in more odd-sized components than $|S|$. Tutte's theorem shows that the converse is true, providing a complete characterization of graphs with perfect matchings.

```
In[84]:= Map[Length, ConnectedComponents[ DeleteVertices[
              NoPerfectMatchingGraph,{6}] ] ]
Out[84]= {5, 5, 5}
```

■ 8.4.2 Bipartite Matching

The study of matchings in graphs arose from the *marriage problem*, where each of b boys knows some subset of g girls. It asks under what conditions the boys can be married so each of them gets a

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

girl that he knows. In graph-theoretic terms, this asks if there is a matching in the bipartite graph $G = (X, Y, E)$ that matches each vertex in X to one in Y. Hall's marriage theorem [Hal35] states that there is a matching in which every boy can be married if and only if every subset S of boys knows a subset of girls at least as large as $|S|$. This criterion provides a way to test whether such a marriage is possible and also to construct one, albeit in exponential time.

A polynomial-time matching algorithm follows from Berge's theorem [Ber57], which states that a matching is maximum if and only if it contains no *augmenting path*. For matching M in a graph G, an *M-alternating path* is a simple path whose edges are alternately matched and unmatched. Any vertex incident on some edge in M is said to be *saturated*. An *M-augmenting path* is an M-alternating path that starts and ends at unsaturated vertices. Berge's theorem suggests the following algorithm: Improve an arbitrary matching M by finding an M-augmenting path P and replacing M with the symmetric difference $(M - P) \cup (P - M)$. This is the matching obtained by flipping the status of the edges in P and is one larger than the old matching. The matching must be maximum when it contains no augmenting path.

The following function implements a breadth-first traversal of G to find M-alternating paths starting from vertices in S.

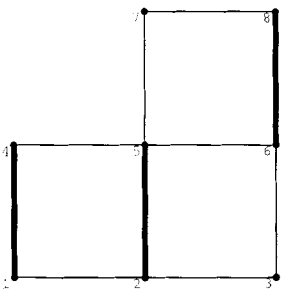
```
AlternatingPaths[g_Graph, start_List, ME_List] :=
     Module[{MV = Table[0, {V[g]}], e = ToAdjacencyLists[g],
            lvl = Table[Infinity, {V[g]}], cnt = 1,
            queue = start, parent = Table[i, {i, V[g]}]},
            Scan[(MV[[#[[1]]]] = #[[2]]; MV[[#[[2]]]] = #[[1]]) &, ME];
            Scan[(lvl[[#]] = 0) &, start];
            While[queue != {},
                 {v, queue} = {First[queue], Rest[queue]};
                 If[EvenQ[lvl[[v]]],
                   Scan[(If[lvl[[#]] == Infinity, lvl[[#]] = lvl[[v]] + 1;
                        parent[[#]] = v; AppendTo[queue, #]]) &, e[[v]]
                   ],
                   If[MV[[v]] != 0,
                     u = MV[[v]];
                     If[lvl[[u]] == Infinity,
                       lvl[[u]] = lvl[[v]] + 1;
                       parent[[u]] = v;
                       AppendTo[queue, u]
                     ]
                   ]
                 ]
            ];
            parent
     ]
```

Finding Augmenting Paths

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

The matching shown in this graph is maximal, but not maximum. The sequence $\{7,5,2,3\}$ is an augmenting path that can improve the matching.

```
In[85]:= g = DeleteVertices[GridGraph[3, 3], {7}];
         m = {{1, 4}, {2, 5}, {6, 8}};
         ShowGraph[Highlight[g, {m}], VertexNumber -> True]
```



Here we compute a tree of alternating paths emanating from 3. The tree is represented by parent pointers – so the parent of vertex 1 is 4, the parent of vertex 2 is 3, and so on. The path $\{7,5,2,3\}$ is represented because 7 points to 5, 5 points to 2, and 2 points to 3. It is an augmenting path because the two endpoints are unsaturated.

```
In[88]:= AlternatingPaths[g, {3}, m]
Out[88]= {4, 3, 3, 5, 2, 3, 5, 6}
```

The *Combinatorica* bipartite matching implementation is based on network flow [HK75]. Take the given bipartite graph G to be an unweighted, directed graph in which every edge goes from the first stage to the second. Add a source s, with edges of unit capacity going from s to each vertex in the first stage, and a sink t, with edges of capacity 1 going from each vertex in the second stage to t. The maximum flow from s to t must correspond to a maximum matching, since it will find a largest set of vertex-disjoint paths, which in this graph consists of disjoint edges from G.

The connection between network flows and augmenting paths works both ways. Finding a path to push flow along in a residual graph is the same problem as finding an M-augmenting path to flip edges on. Thus the Edmonds-Karp network flow algorithm is analogous to the augmenting path algorithm for bipartite matching.

```
BipartiteMatching[g_Graph] :=
      Module[{p,v1,v2,coloring=TwoColoring[g],n=V[g],flow},
            v1 = Flatten[Position[coloring,1]];
            v2 = Flatten[Position[coloring,2]];
            p = BipartiteMatchingFlowGraph[MakeSimple[g],v1,v2];
            Complement[
              Map[Sort[First[#]]&, NetworkFlow[p, n+1, n+2, Edge]],
              Map[{#,n+1}&, v1],
              Map[{#,n+2}&, v2]
            ]
```

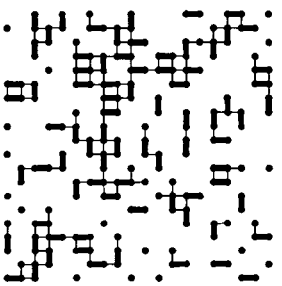
https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

```
      ] /; BipartiteQ[g] && UnweightedQ[g]

BipartiteMatching[g_Graph] := First[BipartiteMatchingAndCover[g]] /; BipartiteQ[g]

BipartiteMatchingFlowGraph[g_Graph, v1_List, v2_List] :=
     Module[{n = V[g], ng},
           ng = ChangeEdges[
                  SetGraphOptions[g, EdgeDirection -> True],
                  Map[If[MemberQ[v1, #[[1]]], #, Reverse[#]] &, Edges[g]]
               ];
           AddEdges[AddVertices[ng, 2],
                  Join[Map[{{n + 1, #}} &, v1], Map[{{#, n + 2}} &, v2]]
           ]
     ]
```

Finding a Maximum Bipartite Matching

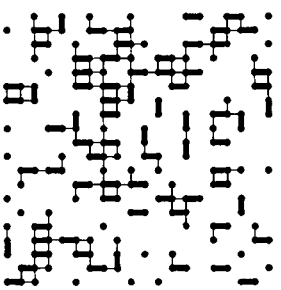
Here is a maximum matching in a random induced subgraph of a grid graph. Roughly half the vertices will be removed from the grid. The matching may leave unsaturated vertices, but there are guaranteed not to be any augmenting paths remaining.

```
In[89]:= g = InduceSubgraph[GridGraph[20, 20], RandomSubset[400]];
         m = BipartiteMatching[g];
         ShowGraph[Highlight[g, {m}]];
```



Here is a *maximal matching* of the same graph. It is maximal in the sense that no edges can be added to it without violating the matching property. However, a maximal matching need not be *maximum*. See if you can find an M-augmenting path for this matching.

```
In[92]:= mm = MaximalMatching[g]; ShowGraph[Highlight[g, {mm}]];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

By definition, there cannot be a matching larger than a maximum matching.

```
In[93]:= {Length[mm], Length[m]}
Out[93]= {72, 79}
```

■ 8.4.3 Weighted Bipartite Matching and Vertex Cover

Weighted bipartite matching seeks a matching in an edge-weighted bipartite graph whose total weight is maximum. There is a remarkable duality between the weighted matching problem and the weighted vertex cover problem, which can be exploited to produce polynomial-time solutions to both problems on bipartite graphs. In this setting, a *vertex cover* seeks to find the *costs* u_i, $1 \le i \le n$, and v_j, $1 \le j \le n$, such that for each i, j, $u_i + v_j \ge w_{ij}$ and the sum of the costs is minimum. Letting u and v respectively denote the vectors $(u_1, u_2, \ldots, u_n)$ and $(v_1, v_2, \ldots, v_n)$, we call the pair (u, v) a *cover*. The *cost* of the cover (u, v), denoted $c(u, v)$, is simply $\sum_i u_i + \sum_j v_j$, and this is what we seek to minimize.

Without loss of generality, we assume that the input graph is the complete weighted bipartite graph $K_{n,n}$. Why complete? Because if it is not we can add vertices to make the two stages of equal size and set all the missing edges to have weight 0 without changing the total weight. We assume the vertices in each of the two stages are labeled 1 through n, and that w_{ij} represents the weight of the edge connecting vertex i to vertex j.

Let M be a perfect matching of $K_{n,n}$ and let $w(M)$ denote its weight. The "duality theorem" expressing the connection between the maximum weight matching problem and the minimum cost vertex cover problem is as follows:

> For any vertex cover (u, v) and any perfect matching M, $c(u, v) \ge w(M)$. Furthermore, $c(u, v) = w(M)$ if and only if every edge $\{i, j\}$ in M satisfies $u_i + v_j = w_{ij}$. In this case, M is a maximum matching and (u, v) is a minimum vertex cover.

Kuhn proposed the *Hungarian algorithm* to exploit this duality, naming it to honor the contributions of Konig and Egervary. The Hungarian algorithm provides a polynomial-time solution to both of these problems. Suppose we have a feasible cover (u, v), that is, a cover (u, v) satisfying $u_i + v_j \ge w_{ij}$ for all edges $\{i, j\}$. By the duality theorem, if this cover is optimal, then we should be able to find a matching whose weight is identical to the cost of the cover. Furthermore, the duality theorem tells us that such a matching can be found among the edges that are "tightly" covered, that is, those edges $\{i, j\}$ for which $u_i + w_j = w_{ij}$. We construct the *equality subgraph* $G_{u,v}$, a spanning subgraph of $K_{n,n}$ whose edges $\{i, j\}$ are all those that satisfy $u_i + w_j = w_{ij}$. Now construct a maximum matching M of $G_{u,v}$. If M is a perfect matching of $K_{n,n}$, we are done, because in this case $c(u, v) = w(M)$.

If not, M can be improved. Here is how. Let X and Y be the vertices in the two stages of $K_{n,n}$. Let U be the unmatched vertices in X, S be the vertices in X reachable by M-alternating paths from U, and T be the vertices in Y reachable by M-alternating paths from U. To increase the size of M, we attempt to bring more edges into $G_{u,v}$ by changing the cover (u, v). First note that edges from S to $Y - T$ are not in $G_{u,v}$. After all, if there is an edge $\{i, j\}$, $i \in S$, $j \in Y - T$ in $G_{u,v}$, what prevents j from being in T? Therefore, for any edge $\{i, j\}$, $i \in S$, and $j \in Y - T$, $u_i + v_j > w_{i,j}$. Now define $\epsilon = \min\{u_i + v_j - w_{i,j} \mid i \in S, j \in Y - T\}$ and reduce u_i by ϵ for all $i \in S$ and increase v_j by ϵ for all

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

$j \in T$. This implies that the sum $u_i + v_j$ remains unaffected for edges between S and T, increases for edges between $X - S$ and T, remains unaffected for edges between $X - S$ and $Y - T$, and decreases just the right amount for edges between S and $Y - T$. This guarantees that $G_{u,v}$ loses no edges and acquires at least one new edge. Now we find a maximum matching in the new graph $G_{u,v}$. This leads to an iterative improvement in M until it becomes a perfect matching of $K_{n,n}$. This algorithm is implemented in the *Combinatorica* function `BipartiteMatchingAndCover`.

We give an edge-weighted $K_{6,6}$ as input to `BipartiteMatchingAndCover`, which returns a maximum matching along with a minimum vertex cover.

```
In[94]:= g = SetEdgeWeights[CompleteGraph[6, 6],
            WeightingFunction->RandomInteger, WeightRange->{1,10}];
          {m, c} = BipartiteMatchingAndCover[g]
Out[95]= {{{1, 8}, {2, 7}, {3, 12}, {4, 11}, {5, 10},
     {6, 9}}, {9, 10, 8, 9, 10, 9, 0, 0, 1, 0, 1, 0}}
```

The equality of these two quantities is proof that the matching and the cover computed above are optimal.

```
In[96]:= {Apply[Plus, GetEdgeWeights[g, m]], Apply[Plus, c]}
Out[96]= {57, 57}
```

The unweighted vertex cover problem can be solved for a bipartite graph G by thinking of edges in G as having weight 1 and edges missing from G as having weight 0. Vertices assigned cost 1 appear in the vertex cover. The duality between matching and vertex cover implies that the size of a maximum matching M equals the size of a minimum vertex cover C, so for every edge in M exactly one of its endpoints is in C.

```
In[97]:= g = InduceSubgraph[GridGraph[10, 10], RandomSubset[100]];
          {m, c} = BipartiteMatchingAndCover[g];
          ShowGraph[Highlight[g, {m, Flatten[Position[c, 1]]}]];
```

The graph h is g plus one edge, but this edge is sufficient to make it nonbipartite. Thus `MinimumVertexCover` uses a brute-force algorithm on h but uses the fast Hungarian algorithm for g. The difference is clear.

```
In[100]:= g = GridGraph[5, 5]; h = AddEdges[g, {{2, 6}}];
           {Timing[c=MinimumVertexCover[h];],
               Timing[MinimumVertexCover[g];]}
Out[101]= {{217.52 Second, Null}, {0.81 Second, Null}}
```

■ 8.4.4 Stable Marriages

Not all matching problems are most naturally described in terms of graphs. Perhaps the most amusing example is the *stable marriage problem*.

Given a set of n men and n women, it is desired to marry them off, one man to one woman. As in the real world, each man has an opinion of each woman and ranks them in terms of desirability from

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

1 to n. The women do the same to the men. Now suppose they are all married off, including couples $\{m_1, w_1\}$ and $\{m_2, w_2\}$. If m_1 prefers w_2 to w_1 and w_2 prefers m_1 to her current spouse m_2, domestic bliss is doomed. Such a marriage is *unstable* because m_1 and w_2 would run off to be with each other. The goal of the stable marriage problem is to find a way to match men and women subject to their preference functions, such that the matching is stable. Obviously, stability is a desirable property, but can it always be achieved?

Gale and Shapely [GS62] proved that, for *any* set of preference functions, a stable marriage exists. Even if one person is so unattractive that everyone ranks that person last, he or she can be assigned a spouse s undesirable enough to all others that no one would be willing to give up their spouse to rescue s. The proof is algorithmic. Starting from his favorite, each unmarried men take turns proposing to the highest rated woman he has not yet proposed to. If a woman gets more than one proposal, she takes the best one, leaving the loser unmarried. Eventually, everyone gets married, since a woman cannot turn down a proposal unless she has a better one, and further, this marriage is stable, since each man always proposes to the highest ranked woman who hasn't rejected him yet. Thus no man can better his lot with further proposals.

The Gale-Shapely algorithm is used in matching hospitals to interns and has led to a well-developed theory of stable marriage [GI89].

```
StableMarriage[mpref_List,fpref_List] :=
      Module[{n=Length[mpref],freemen,cur,i,w,husband},
            freemen = Range[n];
            cur = Table[1,{n}];
            husband = Table[n+1,{n}];
            While[ freemen != {},
                  {i,freemen}={First[freemen],Rest[freemen]};
                  w = mpref[[ i,cur[[i]] ]];
                  If[BeforeQ[ fpref[[w]], i, husband[[w]] ],
                        If[husband[[w]] != n+1,
                              AppendTo[freemen,husband[[w]] ]
                        ];
                        husband[[w]] = i,
                        cur[[i]]++;
                        AppendTo[freemen,i]
                  ];
            ];
            InversePermutation[ husband ]
      ] /; Length[mpref] == Length[fpref]
```

The Gale-Shapely Algorithm for Stable Marriages

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

The Gale-Shapely algorithm finds a stable marriage for any set of preference functions. The ith element of the returned permutation is the wife of man i.

```
In[102]:= TableForm[{ men=Table[RandomPermutation[9],{9}],
          women=Table[RandomPermutation[9],{9}]}]

Out[102]//TableForm=
```

7	4	8	7	3	4	6	7	2
9	5	7	6	2	8	4	3	7
8	1	9	1	5	2	3	8	3
1	6	6	9	1	5	2	4	9
6	8	1	5	4	1	5	9	8
2	3	3	2	6	3	7	2	1
3	2	4	3	9	9	8	5	5
4	7	5	4	7	6	9	1	4
5	9	2	8	8	7	1	6	6

3	8	5	9	1	2	4	8	5
8	1	4	8	8	8	7	7	4
4	3	3	1	9	6	8	5	7
9	4	1	3	2	4	6	9	2
2	2	2	5	3	1	2	3	1
7	6	8	2	4	7	5	2	8
1	7	7	7	5	3	9	6	6
6	9	9	4	7	5	1	4	3
5	5	6	6	6	9	3	1	9

Because of the proposal sequence, the Gale-Shapely algorithm yields the *male-optimal* marriage, under which each man gets his best possible match in any stable marriage.

```
In[103]:= StableMarriage[men,women]
Out[103]= {9, 4, 1, 7, 3, 2, 6, 8, 5}
```

The sexual bias can be reversed by simply exchanging the roles of men and women. The inverse permutation returns who the men are married to. It can be verified that in all couples that differ from the previous matching, the men are less well off and the women happier.

```
In[104]:= InversePermutation[ StableMarriage[women,men] ]
Out[104]= {2, 4, 1, 7, 3, 6, 9, 8, 5}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.5 Partial Orders

In this section, we consider several important problems on directed acyclic graphs (DAGs) and their close cousins, partial orders. A *partially ordered set* (or *poset*) is a set with a consistent ordering relation over its elements. More formally, a partial order is a binary relation that is reflexive, transitive, and antisymmetric. Interesting binary relations between combinatorial objects often are partial orders.

```
PartialOrderQ[r_?SquareMatrix] := ReflexiveQ[r] && AntiSymmetricQ[r] && TransitiveQ[r]
PartialOrderQ[g_Graph] := ReflexiveQ[g] && AntiSymmetricQ[g] && TransitiveQ[g]
```

Partial Order Predicates

■ 8.5.1 Topological Sorting

A directed acyclic graph defines a precedence relation on the vertices, if arc (i, j) is taken as meaning that vertex i must occur before vertex j. A *topological sort* is a permutation p of the vertices of a graph such that an edge (i, j) implies that i appears before j in p.

Only directed acyclic graphs can be topologically sorted, since no vertex in a directed cycle can take precedence over all the rest. Every acyclic graph contains at least one vertex v of out-degree 0. Clearly, v can appear last in the topological ordering. Deleting v leaves a graph with at least one other vertex of out-degree 0. Repeating this argument gives an algorithm for topologically sorting any directed acyclic graph.

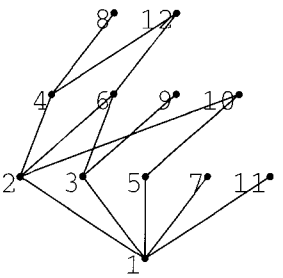
```
TopologicalSort[g_Graph] := Range[V[g]] /; EmptyQ[g]
TopologicalSort[g_Graph] :=
      Module[{g1 = RemoveSelfLoops[g],e,indeg,zeros,v},
            e=ToAdjacencyLists[g1];
            indeg=InDegree[g1];
            zeros = Flatten[ Position[indeg, 0] ];
            Table[{v,zeros}={First[zeros],Rest[zeros]};
                  Scan[(indeg[[#]]--; If[indeg[[#]]==0, AppendTo[zeros,#]])&, e[[v]]];
                  v,
                  {V[g]}
            ]
      ] /; AcyclicQ[RemoveSelfLoops[g]] && !UndirectedQ[g]
```

Topologically Sorting a Graph

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

The divisibility relation induces a partial order on positive integers. More precisely, $i < j$ if and only if i divides j. The Hasse diagram (see Section 8.5.3) of this poset is shown here. Each relation $i < j$ is represented by an "upward path" from i to j in the diagram. Since there is an upward path from 1 to everything else, every topological sort of this poset starts with 1. Any listing of the integers in nondecreasing order of their ranks is a topological sort.

```
In[105]:= ShowGraph[HasseDiagram[g = MakeGraph[Range[12],
          (Mod[#2, #1] == 0) &]], VertexNumber -> True,
          TextStyle->{FontSize->11}];
```



Such a "ranked" topological ordering on the elements is produced here.

```
In[106]:= TopologicalSort[g]
Out[106]= {1, 2, 3, 5, 7, 11, 4, 6, 9, 10, 8, 12}
```

Any permutation of the vertices of an empty graph defines a topological order.

```
In[107]:= TopologicalSort[ EmptyGraph[10] ]
Out[107]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

On the other hand, a complete directed acyclic graph defines a total order, so there is only one possible output from `TopologicalSort`.

```
In[108]:= TopologicalSort[ MakeGraph[Range[10],(#1 > #2)&] ]
Out[108]= {10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

■ 8.5.2 Transitive Closure and Reduction

A directed graph is *transitive* if its edges represent a transitive relation. More precisely, $G = (V, E)$ is transitive if, for any three vertices $x, y, z \in V$, the existence of edges (x, y), $(y, z) \in E$ implies that $(x, z) \in E$. We extend this definition to undirected graphs by interpreting each undirected edge as two directed edges headed in opposite directions.

Any graph G can be made transitive by adding enough extra edges. We define the *transitive closure* $C(G)$ of G as the graph that contains an edge $(u, v) \in C(G)$ if and only if there is a directed path from u to v in G. This definition translates into an $O(n^3)$ algorithm, where we start with the adjacency matrix of G and update it repeatedly to ensure that if slot $[i, j]$ and slot $[j, k]$ are nonzero, then so is slot $[i, k]$.

This algorithm is a perfect candidate for compilation. It consists of three simple nested loops that manipulate a square integer matrix, and therefore it contains no *Mathematica* features that are an obstacle to compilation. In the implementation below, function `TC` is compiled while the wrapper function `TransitiveClosure` converts the graph into an adjacency matrix and passes it along to `TC`.

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

```
TransitiveClosure[g_Graph] := g /; EmptyQ[g]

TransitiveClosure[g_Graph] :=
      Module[{e = ToAdjacencyMatrix[g]},
            If[UndirectedQ[g],
              FromAdjacencyMatrix[TC[e], Vertices[g, All]],
              FromAdjacencyMatrix[TC[e], Vertices[g, All], Type -> Directed]
            ]
      ]

TC = Compile[{{e, _Integer, 2}},
          Module[{ne = e, n = Length[e], i, j, k},
               Do[If[ne[[j, i]] != 0,
                    Do[If[ne[[i, k]] != 0, ne[[j, k]] = 1], {k, n}]
                  ], {i, n}, {j, n}
               ];
               ne
          ]
    ]
```

Finding the Transitive Closure of a Graph

A *transitive reduction* of a graph G is a smallest graph $R(G)$ such that $C(G) = C(R(G))$. Determining a transitive reduction of a binary relation is more difficult, since arcs that were not part of the relation can be in the reduction. Although it is hard to find a smallest subset of the *arcs* determining the relation, Aho, Garey, and Ullman [AGU72] give an efficient algorithm to determine a smallest subset of *vertex pairs* determining the relation. All of these complications go away when the graph is acyclic, as is true with a partial order. In this case, any edge of G that would have been added in finding the transitive closure cannot appear in the transitive reduction. Thus any edge implied by other edges in the graph can simply be deleted. For graphs with directed cycles, this policy might delete every edge in a directed cycle. To get around this problem, we modify the algorithm so that it deletes only edges that are implied by two edges in the current transitive reduction instead of the original graph. This gives *a* reduction, but one that is not necessarily minimal. Like transitive closure, this algorithm is a perfect candidate for compilation.

```
TransitiveReduction[g_Graph] := g /; EmptyQ[g]

TransitiveReduction[g_Graph] :=
      Module[{closure = ToAdjacencyMatrix[g]},
            If[UndirectedQ[g],
              FromAdjacencyMatrix[TR[closure], Vertices[g, All]],
              If[AcyclicQ[RemoveSelfLoops[g]],
                FromAdjacencyMatrix[TRAcyclic[closure], Vertices[g, All], Type->Directed],
                FromAdjacencyMatrix[TR[closure], Vertices[g, All], Type->Directed]
              ]
            ]
```

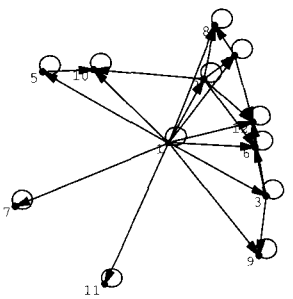
https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

```
      ]

TR = Compile[{{closure, _Integer, 2}},
      Module[{reduction = closure, n = Length[closure], i, j, k},
            Do[
              If[reduction[[i,j]]!=0 && reduction[[j,k]]!=0 &&
                reduction[[i,k]]!=0 && (i!=j) && (j!=k) && (i!=k),
                reduction[[i,k]] = 0
              ],
              {i,n},{j,n},{k,n}
             ];
            reduction
      ]
    ]
```

Finding a Transitive Reduction of an Acyclic Graph

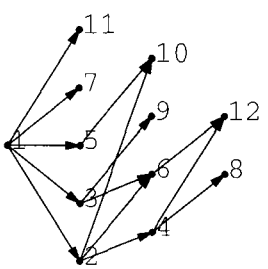
This graph shows the divisibility relation between positive integers as a directed graph. This relation is a partial order, and hence this graph is acyclic modulo self-loops. Since 1 divides everyone else, there is an edge from 1 to every other vertex.

In[109]:=
```
g = MakeGraph[Range[12], (Mod[#2, #1] == 0) &];
ShowGraph[g=Nest[SpringEmbedding,g,10], VertexNumber->True];
```



Here we see the TransitiveReduction of the above graph, which eliminates all implied edges in the divisibility relation, such as 4 | 8, 1 | 4, 1 | 6, and 1 | 8.

In[111]:=
```
ShowGraph[RemoveSelfLoops[ h=RankedEmbedding[
TransitiveReduction[g], {1}]], VertexNumber->True,
VertexNumberPosition->UpperRight, TextStyle->{FontSize->11},
PlotRange -> 0.1];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

Transitive reduction applied to an already reduced graph makes no change to it.

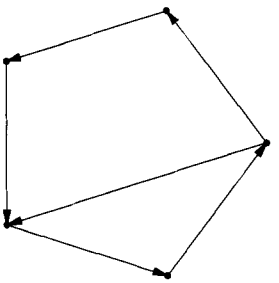
```
In[112]:= IdenticalQ[h, TransitiveReduction[h]]
Out[112]= True
```

The transitive closure of the transitive reduction of a transitive graph is an identity operation.

```
In[113]:= IdenticalQ[g, TransitiveClosure[TransitiveReduction[g]] ]
Out[113]= True
```

The transitive reduction of a nonacyclic graph is not necessarily minimal under our simple reduction algorithm. The directed 5-cycle implies all edges of a complete directed graph. Therefore, we have an unnecessary edge here.

```
In[114]:= ShowGraph[TransitiveReduction[CompleteGraph[5, Type->Directed]]];
```



■ 8.5.3 Hasse Diagrams

What is the best embedding for a partial order? It should clearly identify the hierarchy imposed by the order while containing as few edges as possible to avoid cluttering up the picture. Such a drawing is called a *Hasse diagram* and is the preferred pictorial representation of a partial order.

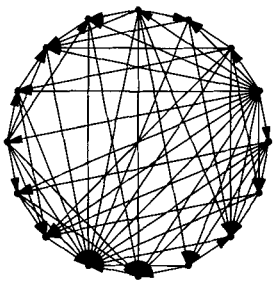
More precisely, the Hasse diagram of a poset P is a drawing with the properties that (i) if $i < j$ in P, then i appears below j and (ii) the drawing contains no edge implied by transitivity. Since all edges in such a drawing go upwards, directions are typically omitted in these drawings.

We have seen how ranked embeddings (Section 5.5.2) can be used to represent a hierarchy. At the bottom of the hierarchy are the vertices that have no ancestors or, equivalently, have in-degree 0. The vertices that have out-degree 0 represent maxima of the partial order and are ranked together at the top. Performing a transitive reduction minimizes the number of edges in the graph. The *Combinatorica* function `HasseDiagram` takes a directed acyclic graph as input and outputs a Hasse diagram.

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

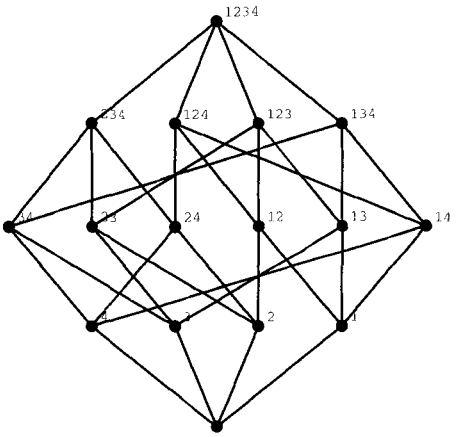
It is impossible to understand the *Boolean algebra*, the partial order on subsets defined by inclusion, from this circular embedding. Beyond the arbitrary position of the vertices, this graph contains too many edges to understand.

```
In[115]:= ShowGraph[s = MakeGraph[Subsets[4],
            ((Intersection[#2,#1]===#1) && (#1 != #2))&]];
```



On the other hand, the Hasse diagram clearly shows the lattice structure, with top and bottom elements. Notice that the vertices of the poset have been placed on horizontal lines or *levels*, with all edges of the Hasse diagram going between vertices in consecutive levels. This makes the Boolean algebra a *ranked poset*, and elements in the same level form a *level set*.

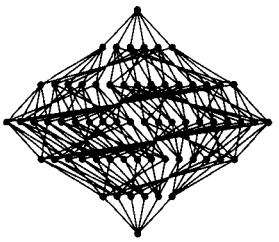
```
In[116]:= l = Map[StringJoin[Map[ToString, #]] &, Subsets[4]];
          ShowGraph[HasseDiagram[s], VertexLabel->l];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

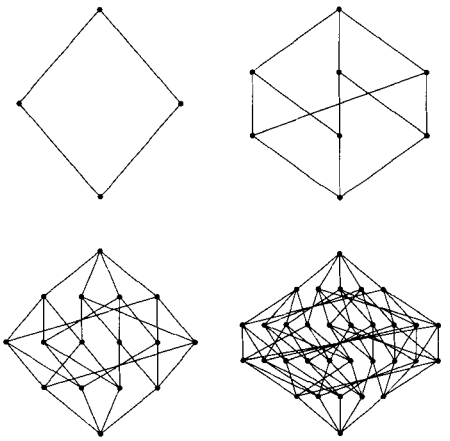
The Boolean algebra and its Hasse diagram are important enough that we provide a function `BooleanAlgebra` to construct a Hasse diagram of this poset.

```
In[118]:= ShowGraph[BooleanAlgebra[6]];
```



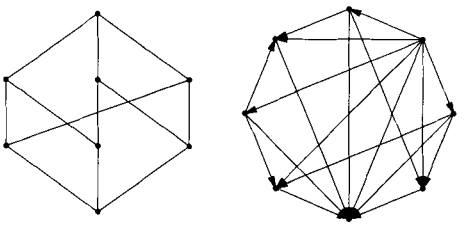
An *antichain* is a set of unrelated elements in a poset. For ranked posets, such as these, it is clear that the level sets are antichains. Can a poset have larger antichains than its largest level set? A finite-ranked poset has the *Sperner property* if the size of a largest antichain equals the size of a largest level set. The Boolean algebra is known to have the Sperner property [SW86].

```
In[119]:= ShowGraphArray[Partition[Table[BooleanAlgebra[n], {n, 2, 5}], 2]];
```



`BooleanAlgebra` takes an option `Type`, which if set to `Directed` produces the underlying directed acyclic graph. While a Hasse diagram is more convenient for visualization, the underlying DAG is more convenient for computations.

```
In[120]:= ShowGraphArray[{BooleanAlgebra[3], BooleanAlgebra[3, Type ->
          Directed]}];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

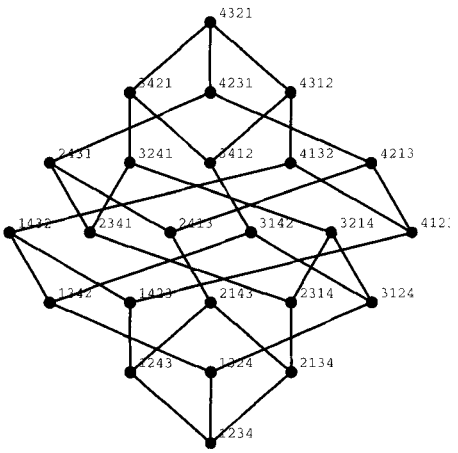
The Boolean algebra is just another fancy embedding for the hypercube!

```
In[121]:= IsomorphicQ[Hypercube[4], BooleanAlgebra[4]]
Out[121]= True
```

The *inversion poset* defines a partial order on n-permutations. For n-permutations π and π', $\pi < \pi'$ if π' can be obtained from π by a sequence of adjacent transpositions, each of which cause a larger element to appear before a smaller one. *Combinatorica* provides a function `InversionPoset` that takes a positive integer n and computes an inversion poset on n-permutations.

As we go up this poset, we go from order to disorder. Despite its seemingly simple structure, no one knows if inversion posets have the Sperner property, in general. In this example, the largest level set is the one in the middle – with size 6. Showing that this poset is Sperner requires showing that it contains no antichain with seven elements.

```
In[122]:= ShowGraph[ InversionPoset[4, VertexLabel -> True]];
```



Another important partial order is the *domination lattice* on integer partitions. Consider integer partitions $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ and $\mu = (\mu_1, \mu_2, \ldots, \mu_j)$ of some positive integer n. We say that λ *dominates* μ if $\lambda_1 + \lambda_2 + \cdots + \lambda_t \geq \mu_1 + \mu_2 + \cdots + \mu_t$ for all $t \geq 1$. We assume that the integer partition with fewer parts is padded with 0's at the end. `DominatingIntegerPartitionQ` tests whether one integer partition dominates another. `DominationLattice` then uses this predicate to construct the partial order on integer partitions and its Hasse diagram.

Since 5 + 1 is smaller than 4 + 3, the first partition does not dominate the second...

```
In[123]:= DominatingIntegerPartitionQ[{5, 1, 1, 1 } , {4, 3, 1}]
Out[123]= False
```

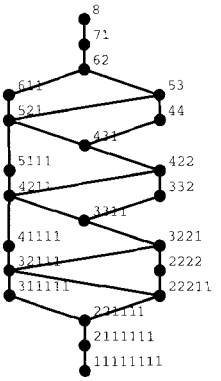
...and since 4 is smaller than 5, the two partitions are incomparable.

```
In[124]:= DominatingIntegerPartitionQ[{4, 3, 1}, {5, 1, 1, 1}]
Out[124]= False
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

This shows the domination lattice on integer partitions of 8. It can be verified by hand that the largest antichain in this poset is of size 2, and hence this poset has the Sperner property. In general, however, domination lattices do not have the Sperner property [SW86].

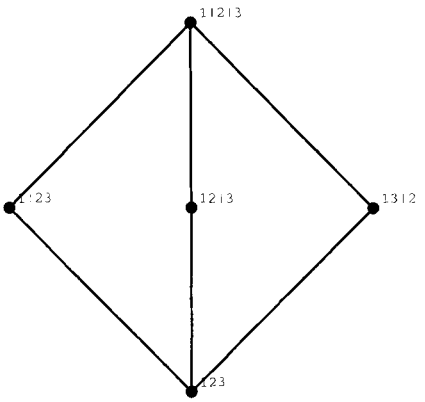
```
In[125]:= ShowGraph[ DominationLattice[8, VertexLabel -> True],
          PlotRange->0.25];
```



The last poset we discuss here is the *partition lattice*, which is a partial order on set partitions. Given two set partitions p and q of a set S, we say $p \leq q$ if p is "coarser" than q. In other words, every block in q is contained in some block in p. *Combinatorica* provides a function `PartitionLattice` that takes a positive integer n and computes a partition lattice on set partitions of $\{1, 2, \ldots, n\}$.

This shows the partition lattice on 3-element set partitions. The bottom element is the coarsest partition, while the top top element is the finest partition possible. The second level (from the bottom) corresponds to set partitions with two blocks and the level above to those with three blocks. The numbers of elements at each level are, therefore, counted by Stirling numbers of the second kind.

```
In[126]:= ShowGraph[PartitionLattice[3, VertexLabel -> True],
          PlotRange -> 0.1];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

Rota [Rot67] conjectured that partition lattices have the Sperner property. Ten years later, Canfield [Can78a] disproved this conjecture using asymptotic methods. No one yet knows the smallest n for which the partition lattice on set partitions of a size n set does not have the Sperner property.

8.5.4 Dilworth's Theorem

A *chain* in a partially ordered set is a collection of elements $v_1, v_2, ..., v_k$ such that v_i is related to v_{i+1}, $i < k$. An *antichain* is a collection of elements no pair of which are related. Dilworth's theorem [Dil50] states that, for any partial order, the maximum size of an antichain equals the minimum number of chains that partition the elements.

To compute a maximum antichain, observe that there is an edge between any two related elements in any transitive relation, such as a partial order. Thus a largest antichain is described by a maximum independent set in the order. To compute a minimum chain partition of the transitive reduction of a partial order G, of order n, we construct a bipartite graph $D(G)$ with two stages of n vertices each, with each vertex v_i of G now associated with vertices $v_{i'}$ and $v_{i''}$. Now each directed edge $\{x, y\}$ of G defines an undirected edge $\{x', y''\}$ of $D(G)$. Each matching of $D(G)$ defines a chain partition of G, since edges $\{x', y''\}$ and $\{y', z''\}$ in the matching represent the chain $\{x, y, z\}$ in G. Further, the maximum matching describes the minimum chain partition.

```
MinimumChainPartition[g_Graph] :=
      ConnectedComponents[
            FromUnorderedPairs[
                  Map[(#-{0,V[g]})&, BipartiteMatching[DilworthGraph[g]]],
                  Vertices[g, All]
            ]
      ]
MaximumAntichain[g_Graph] := MaximumIndependentSet[MakeUndirected[TransitiveClosure[g]]]

DilworthGraph[g_Graph] :=
      FromUnorderedPairs[
            Map[
                  (#+{0,V[g]})&,
                  ToOrderedPairs[RemoveSelfLoops[TransitiveReduction[g]]]
            ]
      ]
```

Partitioning a Partial Order into a Minimum Number of Chains

The is the minimum chain partition of the domination lattice on integer partitions of 8.

```
In[127]:= MinimumChainPartition[d=DominationLattice[8, Type->Directed]]
Out[127]= {{1, 2, 3, 4, 6, 7, 11, 12, 16, 17, 20, 21, 22},
     {5, 8, 9, 10, 13, 14, 15, 18, 19}}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

By Dilworth's theorem, the length of the maximum antichain equals the size of the minimum chain partition for any partial order. This is confirmed by this experiment.

```
In[128]:= antichain = MaximumAntichain[d]
Out[128]= {4, 5}
```

The Hasse diagram of the inversion poset on 4-permutations was shown earlier. From the picture it was clear that 6 is the largest level set in the poset. This shows that the Sperner property is satisfied for this poset.

```
In[129]:= MaximumAntichain[InversionPoset[4, Type->Directed]]
Out[129]= {6, 8, 10, 14, 15, 19}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.6 Graph Isomorphism

Two graphs are *isomorphic* if there exists a renaming of the vertices of the graphs such that they are identical. Two graphs are isomorphic when they have identical structures, although they may be represented differently.

Unfortunately, this notion of equality does not come cheap. There exists no known polynomial-time algorithm for isomorphism testing, despite the fact that isomorphism has not been shown to be NP-complete. It seems to fall in the crack somewhere between P and NP-complete (if such a crack exists), although a polynomial-time algorithm is known when the maximum degree vertex is bounded by a constant [Luk80]. Ironically, an original interest of Leonid Levin, one of two independent developers of the notion of NP-completeness [Lev73], in defining this complexity class was proving that graph isomorphism is hard!

8.6.1 Finding Isomorphisms

We now present an algorithm for finding isomorphisms that, although usually efficient in practice, is not guaranteed to take polynomial time.

An isomorphism between two graphs is described by a one-to-one mapping between the two sets of vertices. Two labeled graphs are *identical* if their current labelings represent an isomorphism.

```
IdenticalQ[g_Graph, h_Graph] := True /; (EmptyQ[g]) && (EmptyQ[h]) && (V[g] == V[h])
IdenticalQ[g_Graph, h_Graph] := False /; (UndirectedQ[g] != UndirectedQ[h])
IdenticalQ[g_Graph, h_Graph] := (V[g]==V[h]) && (Sort[Edges[g]]===Sort[Edges[h]])
```

Testing if Two Graphs are Identical

Every permutation of a complete graph represents a complete graph.

```
In[130]:= Isomorphism[CompleteGraph[10],CompleteGraph[10]]
Out[130]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

These graphs are clearly isomorphic but are not identical.

```
In[131]:= {IdenticalQ[g=CompleteGraph[3,2],h=CompleteGraph[2,3]],
           IsomorphicQ[g,h]}
Out[131]= {False, True}
```

To test for isomorphisms efficiently, we must examine as few of the $n!$ permutations as possible. Using *graph invariants* allows us to prune the number of permutations we must consider, usually to manageable proportions. Graph invariants are measures of a graph that are invariant under isomorphism. Specifically, given a pair of graphs G and H and a vertex v in G, we ask: What vertices in H can map to G in a valid isomorphism? For example, no two vertices of different degrees can be mapped to each other, and so the set of vertices in H that are candidates for being mapped on to v can be pruned by using degree information. Of course, just using vertex degrees may not help

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

at all, but using vertex degrees in conjunction with a few other easily computed graph invariants can significantly help in distinguishing nonisomorphic graphs quickly. The function `Equivalences`, shown below, provides the basic mechanism for pruning the number of candidate permutations that we need to consider.

```
Equivalences[g_Graph, h_Graph, f___] :=
      Module[{dg = Degrees[g], dh = Degrees[h], eq},
            eq = Table[Flatten[Position[dh, dg[[i]]], 1], {i, Length[dg]}];
            EQ[g, h, eq, f]
      ]

EQ[g_Graph, h_Graph, eq_List] := eq
EQ[g_Graph, h_Graph, eq_List, f1_, f___] :=
            If[Position[eq, {}] == {},
              EQ[g, h, RefineEquivalences[eq, g, h, f1], f],
              eq
            ]

Equivalences[g_Graph, f___] := Equivalences[g, g, f]
```

Pruning Candidate Isomorphisms

`Equivalences` uses vertex degrees as the default graph invariant. The output here tells us that either of the two endpoints of the path can be mapped on to the four leaves of the star graph. The central vertex in the star graph is a degree-4 vertex, and there is no degree-4 vertex in a path. This is enough to certify these two graphs as being nonisomorphic.

```
In[132]:= g = Star[5]; h = Path[5]; Equivalences[g, h]
Out[132]= {{1, 5}, {1, 5}, {1, 5}, {1, 5}, {}}
```

Vertex degrees are not useful as an invariant in this case.

```
In[133]:= g = RegularGraph[3, 6]; h = RegularGraph[3, 6];
          Equivalences[g, h]
Out[134]= {{1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6},
     {1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6},
     {1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6}}
```

Combinatorica provides a few functions to compute some other measures of the local structure of a graph. These measures are invariant under graph isomorphism and hence are useful in pruning the isomorphism search space and in distinguishing nonisomorphic graphs.

```
Neighborhood[g_Graph, v_Integer?Positive, 0] := {v} /; (1 <= v) && (v <= V[g])

Neighborhood[g_Graph, v_Integer?Positive, k_Integer?Positive] :=
      Neighborhood[ToAdjacencyLists[g], v, k] /; (1 <= v) && (v <= V[g])
```

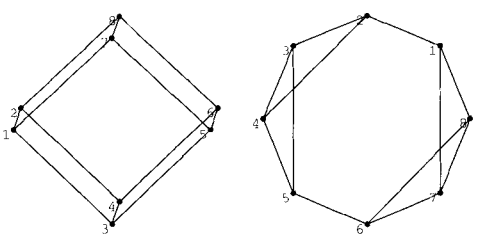
https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

```
Neighborhood[al_List, v_Integer?Positive, 0] := {v} /; (1 <= v)&&(v<=Length[al])
Neighborhood[al_List, v_Integer?Positive, k_Integer?Positive] :=
      Module[{n = {v}},
            Do[n = Union[n, Flatten[al[[ n ]], 1]], {i, k}];
            n
      ] /; (1 <= v) && (v <= Length[al])
DegreesOf2Neighborhood[g_Graph, v_Integer?Positive] :=
       Module[{al = ToAdjacencyLists[g], degrees = Degrees[g]},
             Sort[degrees[[ Neighborhood[al, v, 2] ]]]
       ]

NumberOfKPaths[g_Graph, v_Integer?Positive, 0] := 1 /; (1 <= v) && (v <= V[g])

NumberOfKPaths[g_Graph, v_Integer?Positive, k_Integer?Positive] :=
      NumberOfKPaths[ToAdjacencyLists[g], v, k]

NumberOfKPaths[al_List, v_Integer?Positive, 0] := 1 /; (1<=v)&&(v<=Length[al])
NumberOfKPaths[al_List, v_Integer?Positive, k_Integer?Positive] :=
      Module[{n = {v}},
            Do[n = Flatten[al[[ n ]], 1]  , {i, k}];
            Sort[Map[Length, Split[Sort[n]]]]
      ] /; (1 <= v) && (v <= Length[al])
NumberOf2Paths[g_Graph, v_Integer?Positive] := NumberOfKPaths[g, v, 2]

Distances[g_Graph, v_Integer?Positive] := Sort[BreadthFirstTraversal[g, v, Level]]
```

Some Local Graph Isomorphism Invariants

Here we construct two 3-regular 8-vertex graphs that will form our running example for most of this section. Are these graphs isomorphic? If not, can we distinguish them by using some simple invariants?

```
In[135]:= h = DeleteEdges[CirculantGraph[8, {1, 2}],
            {{2, 8}, {4, 6}, {1, 3}, {5, 7}}];
          g = Hypercube[3]; ShowGraphArray[{g, h}, VertexNumber -> True];
```



Here we use DegreesOf2Neighborhood to compute a sorted list of degrees of vertices in the graph that are within two hops of each vertex. In a three-dimensional hypercube, seven vertices are within two hops of each vertex.

```
In[137]:= Table[DegreesOf2Neighborhood[g, i], {i, V[g]}] // ColumnForm

Out[137]= {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

This graph is not as symmetric as the hypercube. We can get to seven vertices in two hops from some vertices. But we can only get to six vertices in two hops from vertices 3, 4, 7, and 8, and ...

```
In[138]:= Table[DegreesOf2Neighborhood[h, i], {i, V[g]}] // ColumnForm

Out[138]= {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3}
          {3, 3, 3, 3, 3, 3}
```

...this means that none of these four vertices can be mapped onto any vertex in the hypercube. This is enough to distinguish the two graphs.

```
In[139]:= Equivalences[g, h, DegreesOf2Neighborhood]

Out[139]= {{1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6},
    {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6},
    {1, 2, 5, 6}}
```

Each vertex has a path of length 1 to each of its neighbors. A refinement of this is the number of paths of length 2 from a vertex v to various vertices in the graph. These paths need not be simple; here each vertex has three length-2 paths to itself via each of its neighbors. The output here distinguishes the vertex numbered 1 in the two graphs.

```
In[140]:= {NumberOf2Paths[g, 1], NumberOf2Paths[h, 1]}

Out[140]= {{2, 2, 2, 3}, {1, 1, 1, 1, 2, 3}}
```

In fact, using `NumberOf2Paths` as the invariant emphatically distinguishes the two graphs by telling us that no vertex in h can map to any vertex in g.

```
In[141]:= Equivalences[g, h, NumberOf2Paths]

Out[141]= {{}, {}, {}, {}, {}, {}, {}, {}}
```

An invariant that is more difficult to fool is the set of shortest distances between one vertex and all the others [SD76]. For most graphs, this is sufficient to eliminate most nonisomorphic vertex pairs, although there exist nonisomorphic graphs that realize the same set of distances [BH90].

For our example, using `Distances` as an invariant is not as effective as using `NumberOf2Paths`, even though there is some pruning of the search space.

```
In[142]:= Equivalences[g, h, Distances]

Out[142]= {{1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6},
    {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6}, {1, 2, 5, 6},
    {1, 2, 5, 6}}
```

As a starting point for isomorphism testing we use the set of candidate permutations computed by `Equivalences` and refined by applying various invariants. If, at this point, we are already able to distinguish between the graphs, we are done. Otherwise, we may have pruned the search space significantly as preparation for a brute-force backtracking search. We use the general-purpose `Backtrack` function described in Section 7.4.3. Our partial solution will consist of a list of vertices with the property that the ith element belongs to the ith equivalence class. The predicates for testing partial and complete solutions make certain that the induced subgraphs are identical to this point. Other approaches to finding isomorphisms appear in [CG70].

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

```
Isomorphism[g_Graph, h_Graph, equiv_List, flag_Symbol:One] :=
      If[!MemberQ[equiv,{}],
        Backtrack[equiv,
                (IdenticalQ[
                    PermuteSubgraph[g,Range[Length[#]]],
                    PermuteSubgraph[h,#]
                 ] &&
                 !MemberQ[Drop[#,-1],Last[#]]
                )&,
                (IsomorphismQ[g,h,#])&,
                flag
         ],
         {}
      ]
```

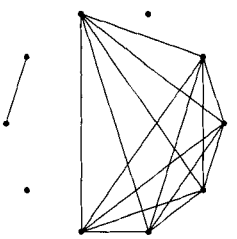
Finding Isomorphisms

We already know that graphs g and h are nonisomorphic, and this is confirmed here.

```
In[143]:= IsomorphicQ[g, h]
Out[143]= False
```

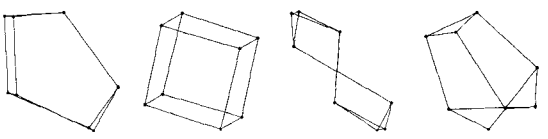
Using our semirandom regular graph generator, we generate ten 8-vertex 3-regular graphs here. We test pairs of these for isomorphism and represent this information in a graph. Vertices of the displayed graph represent 3-regular graphs, and pairs of isomorphic graphs are connected by edges. This graph is a set of disconnected cliques, with each clique representing a set of isomorphic graphs.

```
In[144]:= gt = Table[RegularGraph[3, 8], {10}];
          ShowGraph[h=MakeSimple[MakeGraph[gt, IsomorphicQ, Type ->
          Undirected]]];
```



An independent set in this graph represents a set of mutually nonisomorphic graphs. Here we find a maximum independent set and display the corresponding 3-regular graphs. Do you spot the three-dimensional hypercube here?

```
In[146]:= ShowGraphArray[Map[SpringEmbedding[#, 30]&,
          gt[[MaximumIndependentSet[h]]]]];
```



https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

The functions `IsomorphicQ` and `Isomorphism` take an option `Invariants` that can inform the functions about what invariants to use in pruning the search space. By default, these functions use vertex degrees as the first invariant, followed by `DegreesOf2Neighborhood`, `NumberOf2Paths`, and `Distances`. The order in which these invariants are used may affect the efficiency of the function because the functions return as soon as they can distinguish between the graphs, since there is no point in searching any further.

It is likely that two 3-regular 10-vertex graphs generated using `RegularGraph` are nonisomorphic, and this is usually identified by using the default invariants. The difference in the running times of these two tests comes from the fact that not using invariants forces `IsomorphicQ` to search through many permutations before discovering the futility of it all.

```
In[147]:= g = RegularGraph[3, 10]; h = RegularGraph[3, 10];
          {Timing[IsomorphicQ[g, h];], Timing[IsomorphicQ[g, h,
          Invariants->{}];]}
Out[148]= {{0.15 Second, Null}, {24.59 Second, Null}}
```

We test pairs of random graphs for isomorphism. Usually, the function zips through any pairs of nonisomorphic graphs but will labor over pairs of isomorphic graphs

```
In[149]:= Table[g = RandomGraph[50, .5]; h = RandomGraph[50, .5];
          Timing[ IsomorphicQ[g, h];][[1,1]], {20}]
Out[149]= {0.19, 0.19, 0.19, 0.19, 0.19, 0.19, 0.19, 0.19,
     0.19, 0.19, 0.19, 0.18, 0.19, 0.19, 0.19, 0.18, 0.19,
     0.19, 0.19, 0.19}
```

This gives us an isomorphism ...

```
In[150]:= g = Hypercube[3]; Isomorphism[g, g]
Out[150]= {1, 2, 3, 4, 5, 6, 7, 8}
```

...from among the 48 possible isomorphisms.

```
In[151]:= Length[Isomorphism[g, g, All]]
Out[151]= 48
```

■ 8.6.2 Tree Isomorphism

In the previous section we tested if pairs of graphs are isomorphic by actually attempting to construct an isomorphism between them. Here we present a somewhat different approach, involving the computation of *certificates*, that results in a polynomial-time isomorphism testing algorithm for trees [Rea72]. In general, the fastest graph isomorphism algorithms use the method of computing certificates [KS99].

We compute the certificate of an n-vertex tree T, which is a binary string of length $2n$, as follows. Start with all vertices in T labeled 01. Repeat the following step until two or fewer vertices are left. For each vertex x of T that is not a leaf, let Y be the set of labels of the leaves adjacent to x and the label of x, with the initial 0 and the trailing 1 deleted from the label of x. Replace the label of x with the concatenation of the labels in Y sorted in increasing lexicographic order, with 0 prepended and a 1 appended. Then remove all leaves adjacent to x. After repeating this sufficiently many times, we have one or two vertices left in T. If there is only one vertex x, then the label of x is reported as the certificate. If there are two vertices x and y left, then order these two labels in

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

increasing lexicographic order and report their concatenation as the certificate. This is implemented in the function `TreeToCertificate` and we use this for tree isomorphism testing in `TreeIsomorphismQ`.

The 20-bit certificate of a random 10-vertex tree.

```
In[152]:= TreeToCertificate[RandomTree[10]]
Out[152]= 00001011110000111011
```

Using `TreeIsomorphismQ` is much faster than using `IsomorphicQ` for testing the isomorphism of trees.

```
In[153]:= g = RandomTree[100];
          h = PermuteSubgraph[g, RandomPermutation[V[g]]];
          {Timing[ TreeIsomorphismQ[g, h];],
          Timing[ IsomorphicQ[g, h];]}
Out[155]= {{0.55 Second, Null}, {28.28 Second, Null}}
```

■ 8.6.3 Self-Complementary Graphs

A graph is *self-complementary* if it is isomorphic to its complement.

```
SelfComplementaryQ[g_Graph] := IsomorphicQ[g, GraphComplement[g]]
```

Testing Self-Complementary Graphs

The smallest nontrivial self-complementary graphs are the path on four vertices and the cycle on five.

```
In[156]:= SelfComplementaryQ[ Cycle[5] ] && SelfComplementaryQ[ Path[4] ]
Out[156]= True
```

A simple parity argument shows that every self-complementary graph contains $4k$ or $4k + 1$ vertices.

```
In[157]:= SelfComplementaryQ[ CompleteGraph[3,3] ]
Out[157]= False
```

All self-complementary graphs have diameter 2 or 3 [Sac62].

```
In[158]:= Diameter[ Cycle[5] ]
Out[158]= 2
```

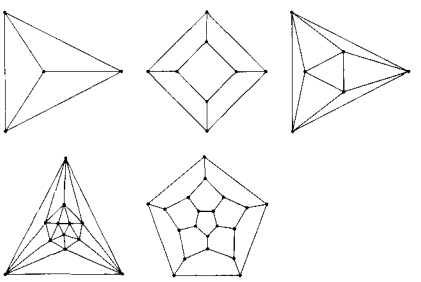
https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.7 Planar Graphs

Planar graphs can be embedded in the plane with no pair of edges crossing. There is an interesting connection between convex polyhedra in E^3 and planar embeddings. For a given polyhedron, replace one of the vertices with a lightbulb, make the faces of the polyhedron out of glass, and make the edges where the faces meet out of lead. These leaded edges cast lines as shadows, which meet at the shadows of the vertices of the polyhedron or, if incident on the light, extend on to infinity. Since the polyhedron is convex, any ray originating from the light passes through exactly one other point on the polyhedron, so no two lines can cross.

The five *Platonic* solids are the convex polyhedra whose faces are all regular d-vertex polygons and whose vertices have the same number of incident edges. They are the *tetrahedron* (four vertices, four triangular faces), the *cube* (eight vertices, six square faces), the *octahedron* (six vertices, eight triangular faces), the *icosahedron* (12 vertices, 20 triangular faces), and the *dodecahedron* (20 vertices, 12 pentagonal faces).

```
In[159]:= ShowGraphArray[gl = {{TetrahedralGraph, CubicalGraph,
            OctahedralGraph}, {IcosahedralGraph, DodecahedralGraph}} ];
```



`PlanarQ` confirms that these graphs are indeed planar.

```
In[160]:= Map[PlanarQ, Flatten[gl]]
Out[160]= {True, True, True, True, True}
```

A polynomial-time algorithm for testing planarity can be obtained from the famous Kuratowski's theorem [Kur30]. Two graphs are *homeomorphic* if they can be obtained from the same graph by replacing edges with paths. Kuratowski's theorem states that a graph is planar if and only if it contains no subgraph homeomorphic to K_5 or $K_{3,3}$. It is easy to verify that K_5 and $K_{3,3}$ have no planar drawings. What is not as easy to see is that these are essentially the only two obstacles to planarity.

Planar graphs have also drawn attention because of the famous *four–color theorem*, which says that the vertices of any planar graph can be colored with at most four colors. It took more than 100 years and many incorrect proofs before this claim was proved by Hakan and Appel [AHK77, AH77]. One of the side effects of this quest was an algorithm by Kempe that produces a 5-coloring of any planar graph. Wagon [Wag98] describes a nice *Mathematica* package built on top of *Combinatorica* that implements Kempe's 5-coloring algorithm and a modification of it that produces 4-colorings on most planar graphs. This package also contains a semirandom algorithm for generating planar graphs and several examples of large planar graphs.

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

■ 8.7.1 Testing Planarity

Although there are several efficient algorithms [CHT90, Eve79, HT74] for testing planarity, all are difficult to implement. Most of these algorithms are based on ideas from an old $O(n^3)$ algorithm by Auslander and Parter [AP61]. Their algorithm is based on the observation that if a graph is planar, for any cycle there is a way to embed it in the plane so that the vertices of this cycle lie on a circle. This follows because an embedding on a sphere is equivalent to an embedding in the plane, as shown above. If this cycle is then deleted from the graph, the connected components that are left are called *bridges*. Obviously, if any bridge is nonplanar, the graph is nonplanar. Further, the graph is nonplanar if these bridges interlock in the wrong way.

By Kuratowski's theorem, neither $K_{3,3}$ nor K_5 is planar.

```
In[161]:= PlanarQ[CompleteGraph[5]] || PlanarQ[CompleteGraph[3,3]]
Out[161]= False
```

Every planar graph on nine vertices has a nonplanar complement [BHK62].

```
In[162]:= PlanarQ[ GraphComplement[GridGraph[3,3]] ]
Out[162]= False
```

All trees are planar. The current embedding has no effect on whether `PlanarQ` rules planar or nonplanar.

```
In[163]:= Apply[And, Map[PlanarQ, Table[RandomTree[i], {i,1,20}] ]]
Out[163]= True
```

The Platonic solids can all be constructed using other *Combinatorica* commands. Here we build the tetrahedron, cube, and octahedron. A proof [Mes83] that there exist no other Platonic solids follows from Euler's formula that for any planar graph with V vertices, E edges, and F faces, $V - E + F = 2$.

```
In[164]:= PlanarQ[CompleteGraph[4]] && PlanarQ[CompleteGraph[2,2,2]] &&
          PlanarQ[GraphProduct[CompleteGraph[2], Cycle[4]]]
Out[164]= True
```

Euler's formula implies that every planar graph has at most $3n - 6$ edges for $n \geq 5$.

```
In[165]:= Apply[Or, Map[PlanarQ, Table[ExactRandomGraph[n,3n-5],
          {n,5,15}]]]
Out[165]= False
```

Coming in under this limit gives us a chance at planarity, at least for small enough n.

```
In[166]:= Map[PlanarQ, Table[ExactRandomGraph[n,3n-6], {n,5,15}]]
Out[166]= {True, True, False, False, False, False, False,
      False, False, False, False}
```

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

8.8 Exercises

8.8.1 Thought Exercises

1. In Section 8.1 we mentioned that the Bellman-Ford algorithm can be used to detect the presence of negative cycles in a graph. Show how the Bellman-Ford algorithm can be extended to return a negative cycle if the given graph contains at least one.
2. Which pair of vertices in the $n \times n$ grid graphs have the largest number of length-k paths between them, as a function of k?
3. Prove that the most expensive edge on any cycle of G will not appear in the minimum spanning tree of G.
4. Prove that the minimum spanning tree also minimizes the cost of the maximum edge (i.e., the "bottleneck" edge) over all such trees.
5. Prove that the center of any tree consists of at most two adjacent vertices.
6. Consider a graph in which each edge has an upper bound and a nonnegative lower bound on the permitted flow through that edge. Show how to determine if such a graph has a *feasible flow*, that is, a flow that satisfies flow conservation at every vertex and flow bounds at each edge. **Hint:** Reduce the problem to the maximum-flow problem.
7. The problem of finding a minimum-size maximal matching is NP-complete, even for planar bipartite graphs [YG80]. Construct a graph that has maximal matchings of many different sizes.
8. Construct a family of graphs such that, for infinitely many positive integers n, there exists a graph in the family with minimum vertex cover size equal to $2n$ and maximum matching size equal to n.
9. The sizes of the level sets of a Boolean algebra are the binomial coefficients. What about the sizes of the level sets of an inversion poset? Have you encountered these combinatorial numbers in an earlier chapter?
10. Determine if the inversion poset on size-4 permutations has the Sperner property.
11. Devise an algorithm that takes a length-$2n$ binary string s and tests if s is a tree certificate and, if so, computes the tree for which s is a certificate.
12. In 1758, Euler proved his famous formula for planar graphs. If a connected graph with n vertices and m edges has a planar embedding with f faces, then $n - m + f = 2$. Use this to derive the fact that every n-vertex planar graph has at most $3n - 6$ edges. Use this upper bound on the number of edges to conclude that every planar graph has a vertex of degree at most 5.
13. Use Euler's formula to show that the cube, tetrahedron, octahedron, dodecahedron, and icosahedron are the only possible Platonic solids.

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press

14. The *dual graph* G^* of a planar graph G is a graph with a vertex for every face in G. For every edge e in G with faces X and Y on either of its sides, there is an edge e^* in G^* connecting the vertices corresponding to faces X and Y. Note that a dual graph is not necessarily simple and may have multiple parallel edges and self-loops. Different planar embeddings of a graph may produce different dual graphs. Show two planar embeddings of a planar graph whose dual graphs are nonisomorphic.
15. An *outerplanar graph* is a planar graph with a planar embedding in which all vertices lie on the single unbounded face. Show that an outerplanar graph has at most $2n - 3$ edges.
16. Show that a graph is outerplanar if and only if it has no subgraph homeomorphic to K_4 or $K_{2,3}$.

■ 8.8.2 Programming Exercises

1. Use the *Mathematica* function `LinearProgramming` to compute the maximum flow through a network by setting up a system of inequalities such that the flow through each edge is bounded by its capacity and the flow into each vertex equals the flow out, while maximizing the flow into the sink. How does the efficiency of this routine compare to `NetworkFlow`?
2. Write a function to find the strongly connected components of a graph using `TransitiveClosure`. How does your function's performance compare to `StronglyConnectedComponents`?
3. Write a function to partition the edges of K_{2n} into n spanning trees.
4. Develop and implement a simple algorithm for finding the maximum matching in a tree, which is special case of bipartite matching. Find a tree for which `MaximalMatching` does not find the maximum matching.
5. Implement Prim's algorithm for finding minimum spanning trees and compare it with `MinimumSpanningTree` for efficiency on a variety of graphs.
6. Write a function to test whether a given matching is stable, for a given pair of male and female preference functions.
7. Rewrite `PlanarQ` so that it positions the vertices of the graph in a way that permits a planar embedding. These positions do not have to permit a straight-line embedding.
8. Implement a function that takes a planar embedding of a graph and returns the faces of the embedding.
9. Use the function in the previous exercise to implement a function that takes the planar embedding of a graph and constructs the dual graph.

8.8.3 Experimental Exercises

1. How does the expected diameter of a complete random graph grow as a function of the number of vertices?
2. Both the domination lattice and the partition lattice do not, in general, have the Sperner property [SW86]. See if you can find what values of n give the smallest non-Sperner lattices for either of them.
3. Experiment with computing shortest-path spanning trees, minimum spanning trees, and traveling salesman tours on random Euclidean graphs. Are these subgraphs always planar embeddings?
4. What is the expected size of the maximum matching in a tree on n vertices?
5. Experiment to determine what graphs are "square roots" [Muk67] of some graph G, meaning a graph whose square is G.
6. The *divorce digraph* [GI89] is a binary relation associated with an instance of the stable marriage problem. The vertices correspond to the $n!$ matchings. There is a directed edge $\{a, b\}$ between matchings a and b if b results from an unstable pair in a leaving their respective spouses and marrying each other, with the rejects getting paired off. Stable marriages are vertices with out-degree 0 in the divorce digraph.
   Write a function to construct the divorce digraph of a stable marriage instance. Verify that the following male $\{\{1,2,3\},\{1,2,3\},\{2,1,3\}\}$ and female $\{\{3,2,1\},\{2,3,1\},\{1,2,3\}\}$ preference functions result in a divorce digraph that contains a directed cycle. Further, show that whatever permutations represent male 1's and female 3's preference functions, the divorce digraph contains a cycle. Does there always exist a directed path from any unstable marriage to a stable one in the divorce digraph of the previous problem [GI89]?
7. Experiment to find pairs of nonisomorphic graphs for which the graph invariants built into `IsomorphicQ` are useless. Are there other simple graph invariants that distinguish between these graphs?

https://doi.org/10.1017/CBO9781139164849.009 Published online by Cambridge University Press