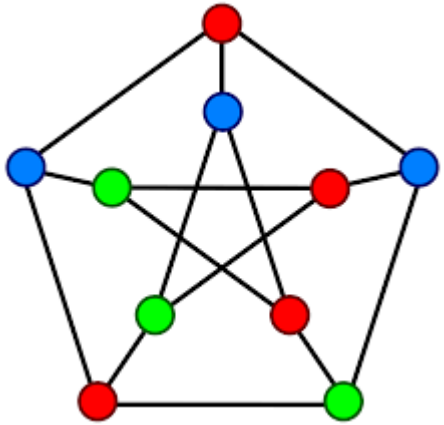
# 10. Graphs and Trees

# 10.1 Graphs: Definitions and Basic Properties

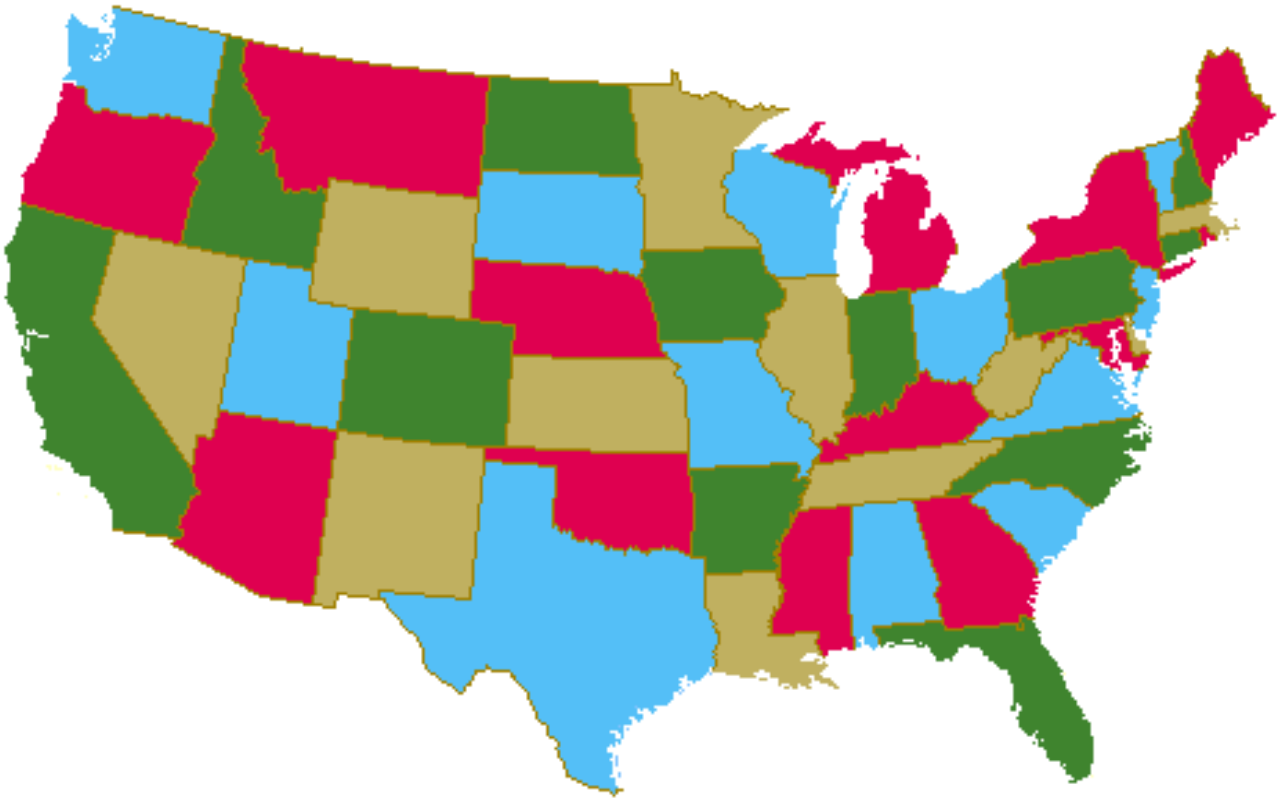


# Graphs: Introduction

Shall we add some colours to this map of the United States?

# Graphs: Introduction

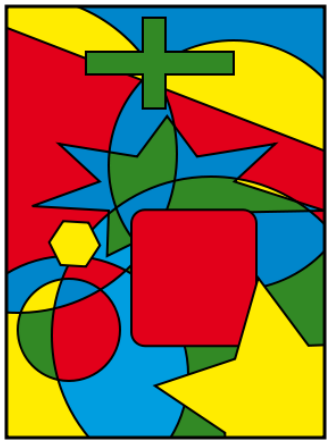
Shall we add some colours to this map of the United States?



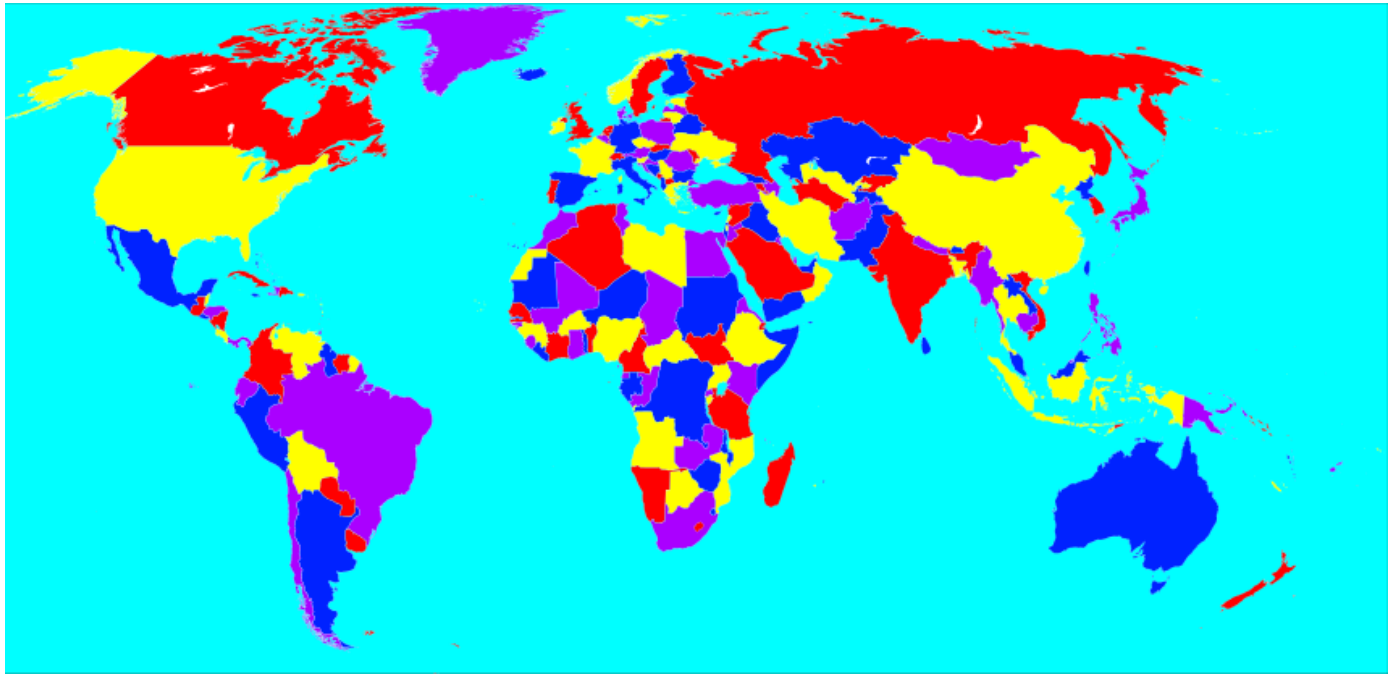
# Map Colouring

- Four-Colour Conjecture
  - Proposed by Guthrie in 1852, who conjectured that…
  - Four colours are sufficient to colour any map in a plane, such that regions that share a common boundary do not share the same colour.
  - Many false proofs since then.
  - Finally proved by Appel and Haken in 1977, with the help of computer.
  - Robertson et al. provided another proof in 1996.

# Map Colouring



Example of a 4-coloured map.



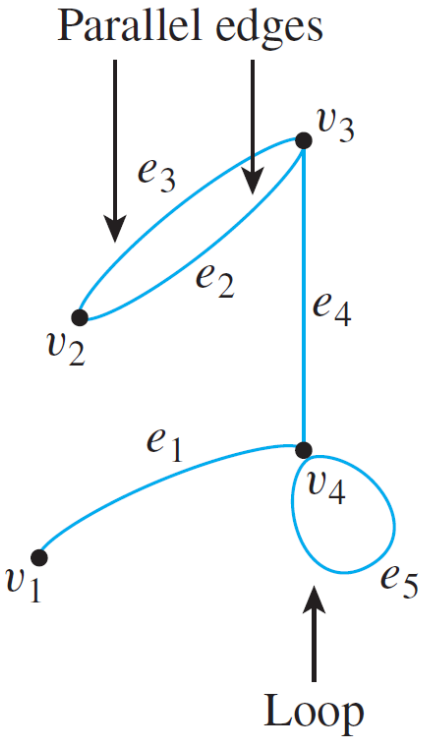
World map with 4 colours.

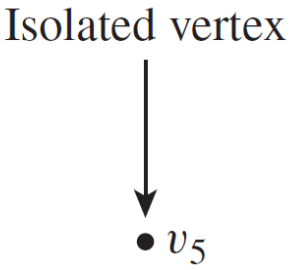
- But this is a map, not a graph!
- However, we can model it as a graph.
- But what is a graph?

# Graphs: Definitions and Basic Properties

- A graph $G$ consists of
  - a set of vertices $V(G)$, and
  - a set of edges $E(G)$.
- Sometimes, we write $G = \{V, E\}$.

An edge connects one vertex to another, or a vertex to itself.




# Graphs: Definitions and Basic Properties

## Definition: Graph

A **graph** $G$ consists of 2 finite sets: a nonempty set $V(G)$ of **vertices** and a set $E(G)$ of **edges**, where each edge is associated with a set consisting of either one or two vertices called its **endpoints**.

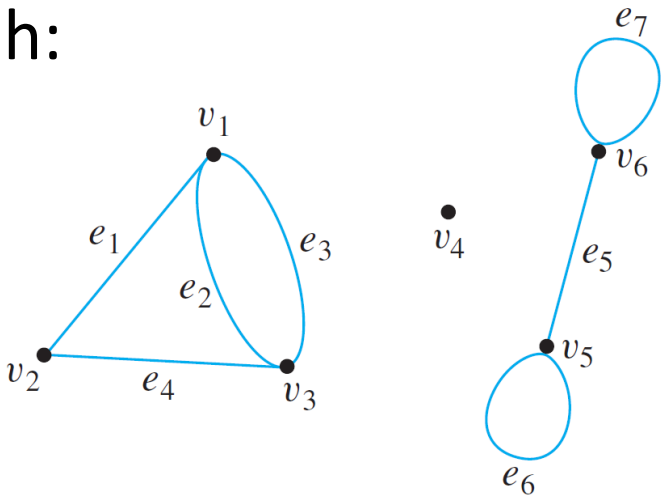
An edge is said to **connect** its endpoints; two vertices that are connected by an edge are called **adjacent vertices**; and a vertex that is an endpoint of a loop is said to be **adjacent to itself**.

An edge is said to be **incident on** each of its endpoints, and two edges incident on the same endpoint are called **adjacent edges**.

We write $e = \{v, w\}$ for an edge $e$ incident on vertices $v$ and $w$.

# Graphs: Definitions and Basic Properties

**Example:** Consider the following graph:




a. Write the vertex set $V$ and the edge set $E$, and give the list of edges with their end-points.

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

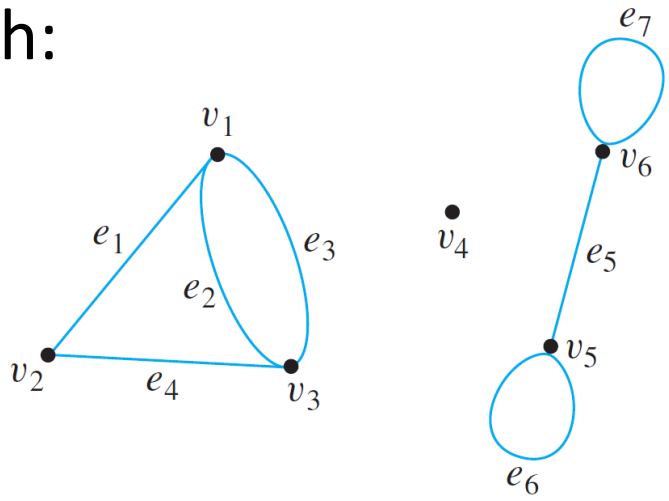
$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$

$e_1 = \{v_1, v_2\}$
$e_2 = \{v_1, v_3\}$
$e_3 = \{v_1, v_3\}$

$e_4 = \{v_2, v_3\}$
$e_5 = \{v_5, v_6\}$
$e_6 = \{v_5\}$
$e_7 = \{v_6\}$

# Graphs: Definitions and Basic Properties

Example: Consider the following graph:




b. Find all edges that are incident on $v_1$, all vertices that are adjacent to $v_1$, all edges that are adjacent to $e_1$, all loops, all parallel edges, all vertices that are adjacent to themselves, and all isolated vertices.

Edges incident on $v_1$: $e_1$, $e_2$ and $e_3$.
Vertices adjacent to $v_1$: $v_2$ and $v_3$.
Edges adjacent to $e_1$: $e_2$, $e_3$ and $e_4$.
Loops: $e_6$ and $e_7$.
$e_2$ and $e_3$ are parallel.

$v_5$ and $v_6$ are adjacent to themselves.
Isolated vertex: $v_4$.

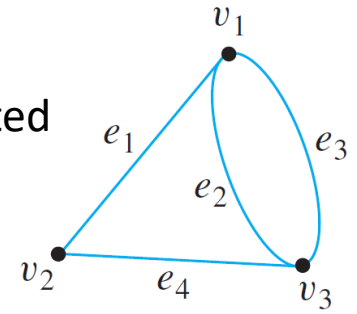
# Graphs: Definitions and Basic Properties

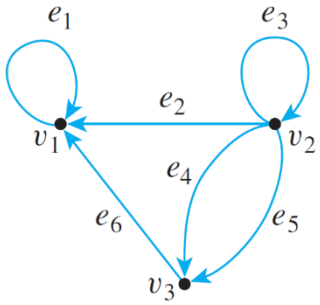
## Definition: Directed Graph

A **directed graph**, or **digraph**, $G$, consists of 2 finite sets: a nonempty set $V(G)$ of **vertices** and a set $D(G)$ of **directed edges**, where each edge is associated with an ordered pair of vertices called its **endpoints**.

If edge $e$ is associated with the pair $(v, w)$ of vertices, then $e$ is said to be the (**directed**) **edge** from $v$ to $w$. We write $e = (v, w)$.

Undirected graph




Directed graph

# Modelling Graph Problems


## Map Colouring Problem

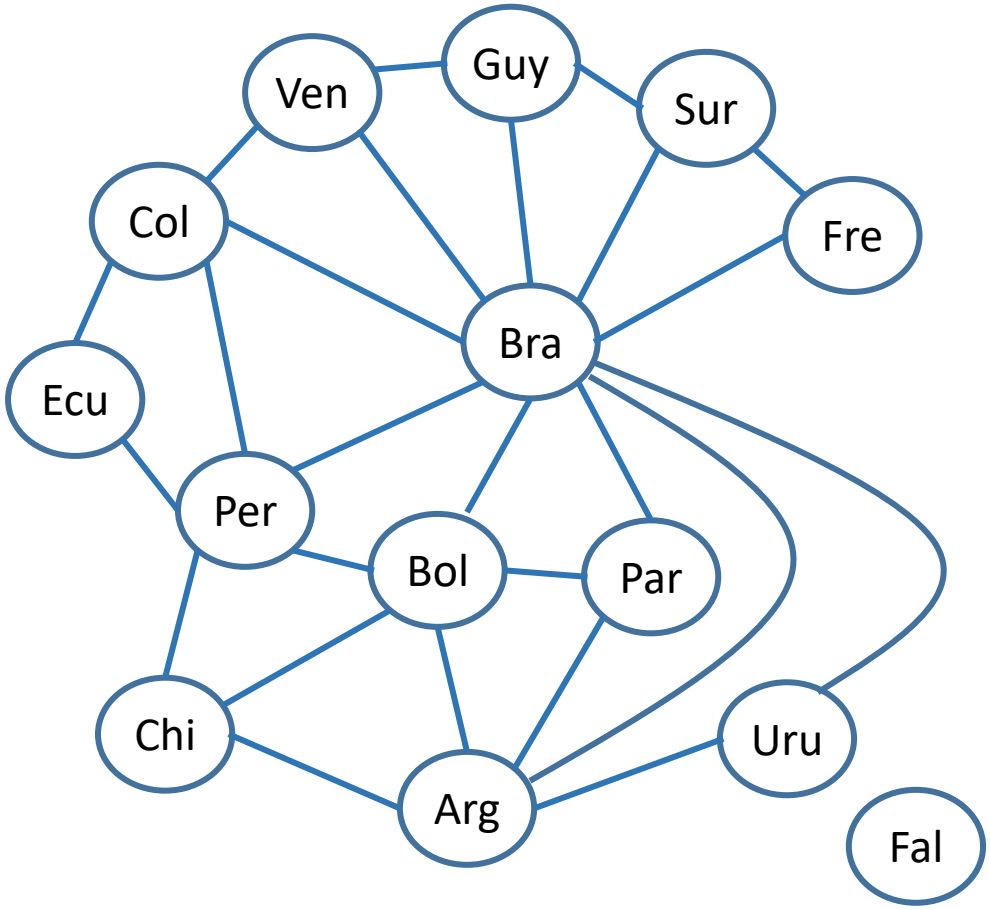
Solve it as a graph problem.

Draw a graph in which the vertices represent the states, with every edge joining two vertices represents the states sharing a common border.

Such two vertices cannot be coloured with the same colour.

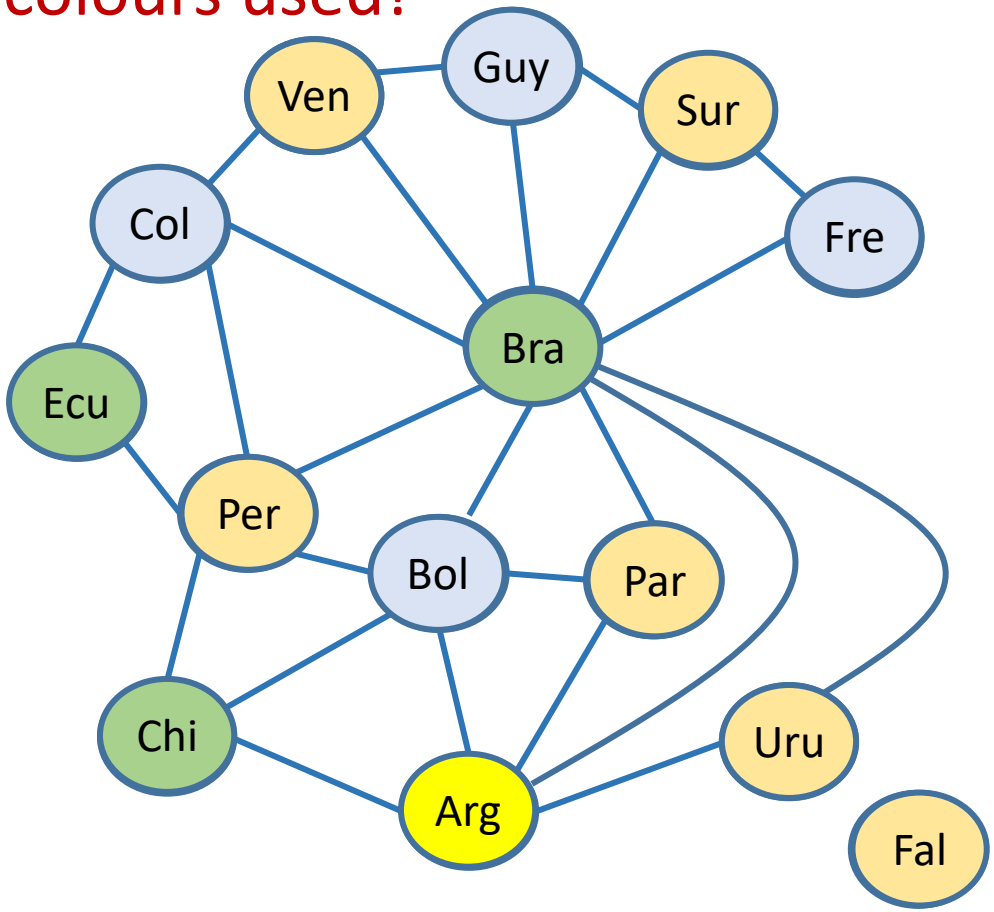
A **vertex colouring** of a graph is an assignment of colours to vertices so that no two adjacent vertices have the same colour.

# Modelling Graph Problems

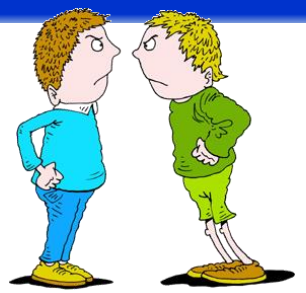



# Modelling Graph Problems


4 colours used!



# Wedding Planner



You are your best friend's wedding planner and you need to plan the seating arrangement for his 16 guests attending his wedding dinner. However, some of the guests cannot get along with some others.

- *A* doesn't get along with *F*, *G* or *H*.
- *B* doesn't get along with *C*, *D* or *H*.
- *C* doesn't get along with *B*, *D*, *E*, *G* or *H*.
- *D* doesn't get along with *B*, *C* or *E*.
- *E* doesn't get along with *C*, *D*, *F*, or *G*.
- *F* doesn't get along with *A*, *E* or *G*.
- *G* doesn't get along with *A*, *C*, *E* or *F*.
- *H* doesn't get along with *A*, *B* or *C*.

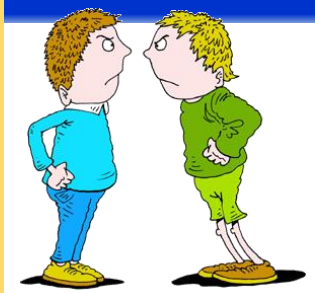
You don't want to put guests who cannot get along with each other at the same table!

How many tables do you need?

Acknowledgement: http://www.math.uri.edu/~eaton/0131873814_MEb.pdf

# Wedding Planner

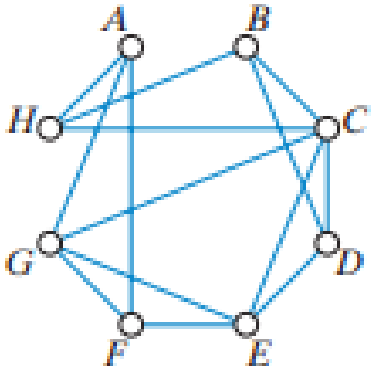
- *A* doesn't get along with *F*, *G* or *H*.
- *B* doesn't get along with *C*, *D* or *H*.
- *C* doesn't get along with *B*, *D*, *E*, *G* or *H*.
- *D* doesn't get along with *B*, *C* or *E*.
- *E* doesn't get along with *C*, *D*, *F*, or *G*.
- *F* doesn't get along with *A*, *E* or *G*.
- *G* doesn't get along with *A*, *C*, *E* or *F*.
- *H* doesn't get along with *A*, *B* or *C*.



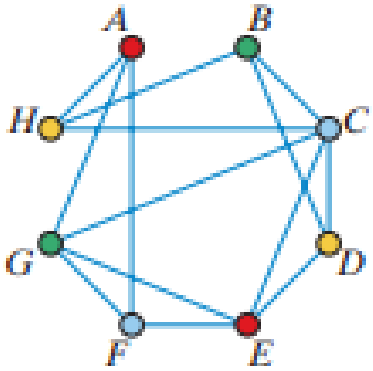
Graph with vertices representing the guests, and an edge is drawn between two guests who don't get along.

Vertex colouring problem.
4 colours (4 tables)?

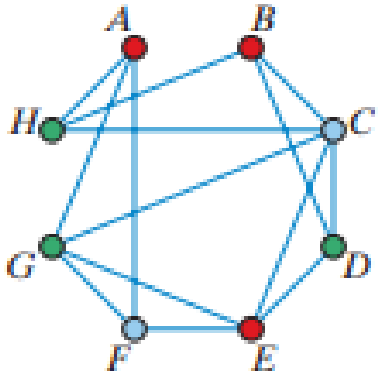
3 colours
(3 tables)!




(a)




(b)




(c)

Acknowledgement: http://www.math.uri.edu/~eaton/0131873814_MEb.pdf

# Simple Graphs

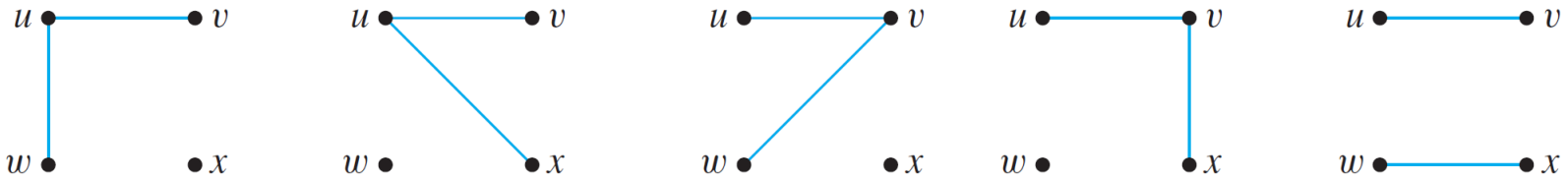
## Definition: Simple Graph

A **simple graph** is an undirected graph that does not have any loops or parallel edges.

Example: Draw all simple graphs with the 4 vertices $\{u, v, w, x\}$ and two edges, one of which is $\{u, v\}$.

There are at most $\binom{4}{2}$ = 6 edges in a simple graph with 4 vertices. One edge is given. Hence we need to pick another from the remaining 5.

Draw the rest.

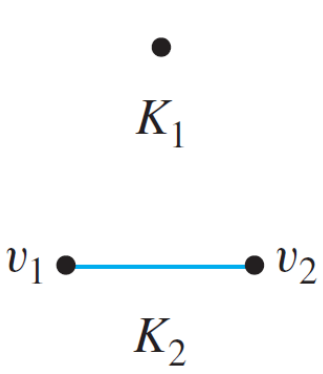


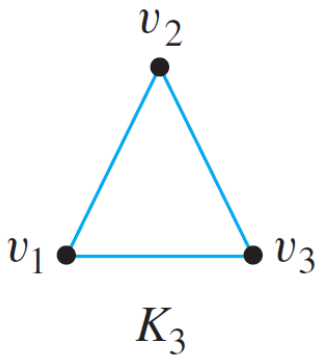
# Complete Graphs

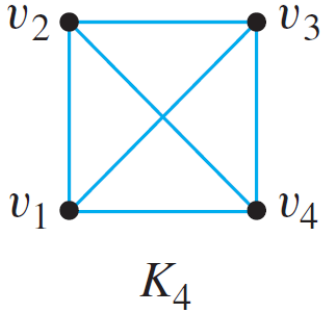
## Definition: Complete Graph

A **complete graph** on ***n* vertices**, $n > 0$, denoted $\boldsymbol{K_n}$, is a simple graph with $n$ vertices and exactly one edge connecting each pair of distinct vertices .

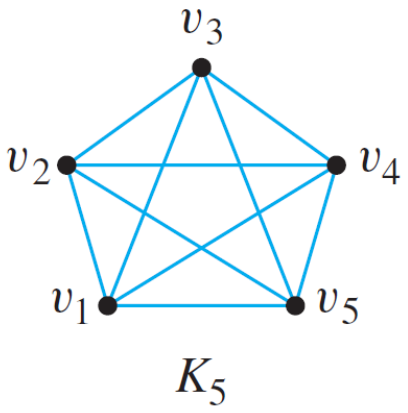
Example: The complete graphs $K_1$, $K_2$, $K_3$ and $K_4$.




Draw $K_5$.



# Complete Bipartite Graphs

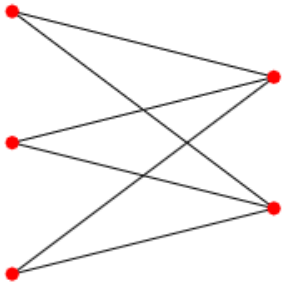
## Definition: Complete Bipartite Graph

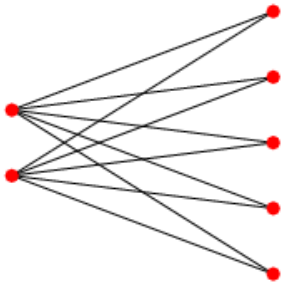
A **complete bipartite graph** on **($m$, $n$) vertices**, where $m, n > 0$, denoted $\boldsymbol{K_{m,n}}$, is a simple graph with distinct vertices $v_1, v_2, \ldots, v_m$, and $w_1, w_2, \ldots, w_n$ that satisfies the following properties:

For all $i, k = 1, 2, \ldots, m$ and for all $j, l = 1, 2, \ldots, n$,

1. There is an edge from each vertex $v_i$ to each vertex $w_j$.
2. There is no edge from any vertex $v_i$ to any other vertex $v_k$.
3. There is no edge from any vertex $w_j$ to any other vertex $w_l$.

$K_{3,2}$





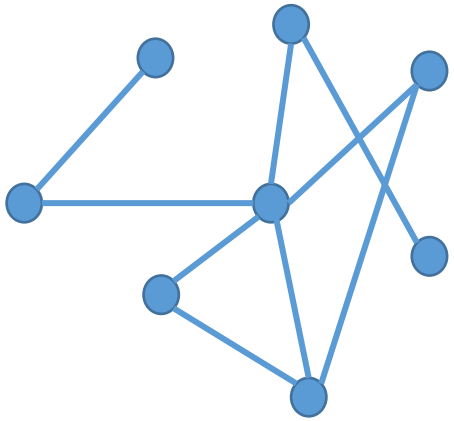
$K_{2,5}$

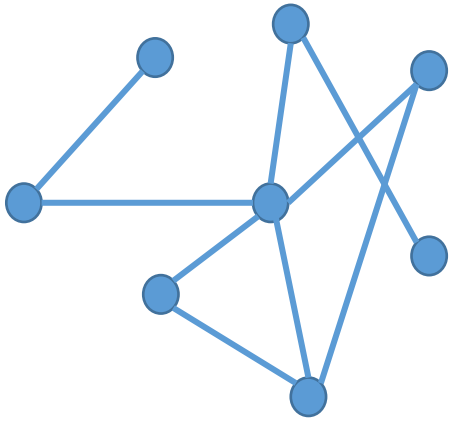
# Subgraph of a Graph

## Definition: Subgraph of a Graph

A graph *H* is said to be a **subgraph** of graph *G* if, and only if, every vertex in *H* is also a vertex in *G*, every edge in *H* is also an edge in *G*, and every edge in *H* has the same endpoints as it has in *G*.

A graph *G*





A subgraph of *G*

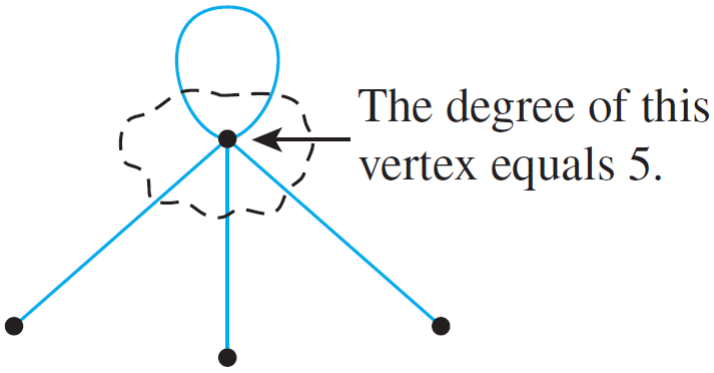
# Degree of a Vertex and Total Degree of a Graph

## Definition: Degree of a Vertex and Total Degree of a Graph

Let *G* be a graph and *v* a vertex of *G*. The **degree** of *v*, denoted **deg(v),** equals the number of edges that are incident on *v*, with an edge that is a loop counted twice.

The **total degree of G** is the sum of the degrees of all the vertices of *G*.

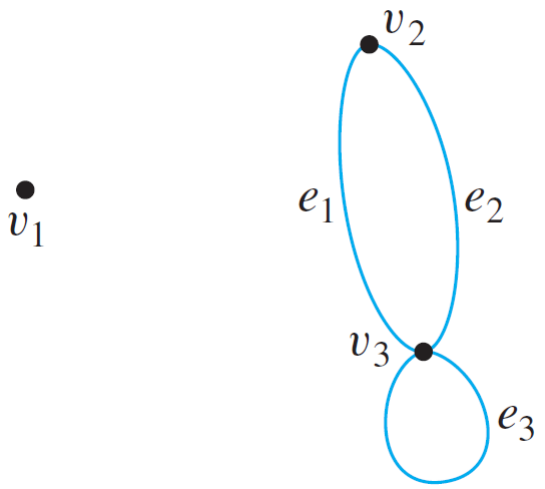
The degree of a vertex can be obtained from the drawing of a graph by counting how many end segments of edges are incident on the vertex.



# Degree of a Vertex and Total Degree of a Graph

Example: Find the degree of each vertex of the graph *G* shown below. Then find the total degree of *G*.

$\deg(v_1) = 0$
$\deg(v_2) = 2$
$\deg(v_3) = 4$
Total degree of $G = 6$



## Theorem 10.1.1 The Handshake Theorem

If $G$ is any graph, then the sum of the degrees of all the vertices of $G$ equals twice the number of edges of $G$. Specifically, if the vertices of $G$ are $v_1, v_2, \ldots, v_n$, where $n \geq 0$, then

$$\text{The total degree of } G = \deg(v_1) + \deg(v_2) + \ldots + \deg(v_n)$$
$$= 2 \times (\text{the number of edges of } G).$$

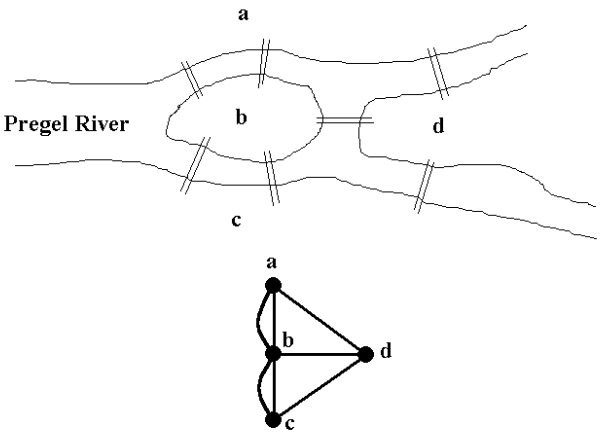
## Corollary 10.1.2

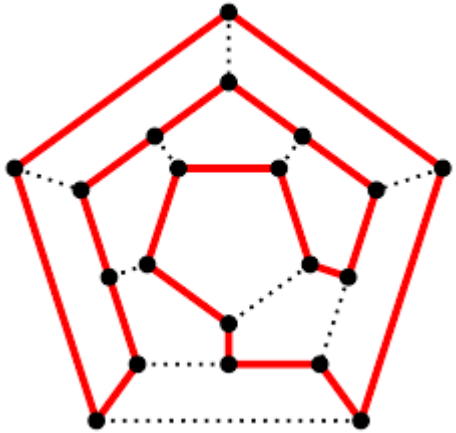
The total degree of a graph is even.

## Proposition 10.1.3

In any graph there are an even number of vertices of odd degree.

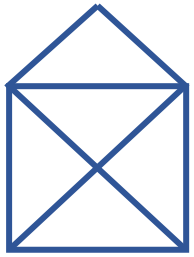
# 10.2 Trails, Paths, and Circuits



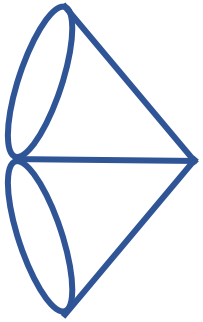



# Let's Have Some Fun

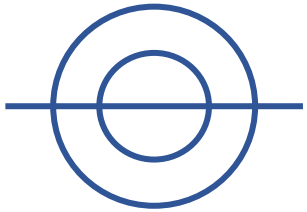
Can you draw the following figures without lifting up your pencil?



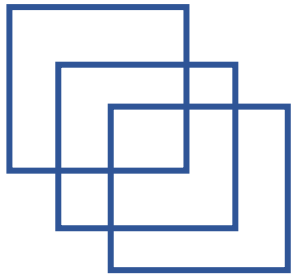
(1)



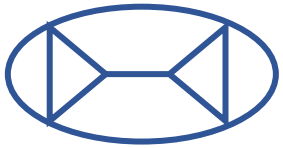
(2)



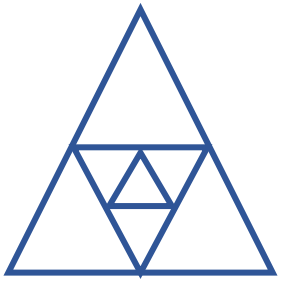
(3)



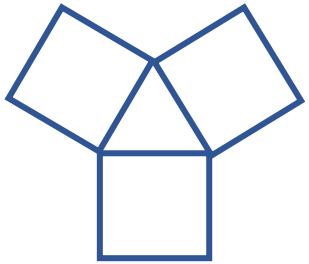
(4)



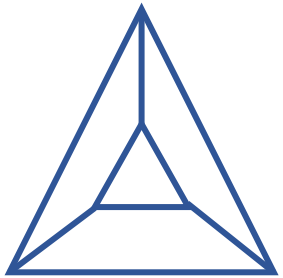
(5)



(6)



(7)



(8)

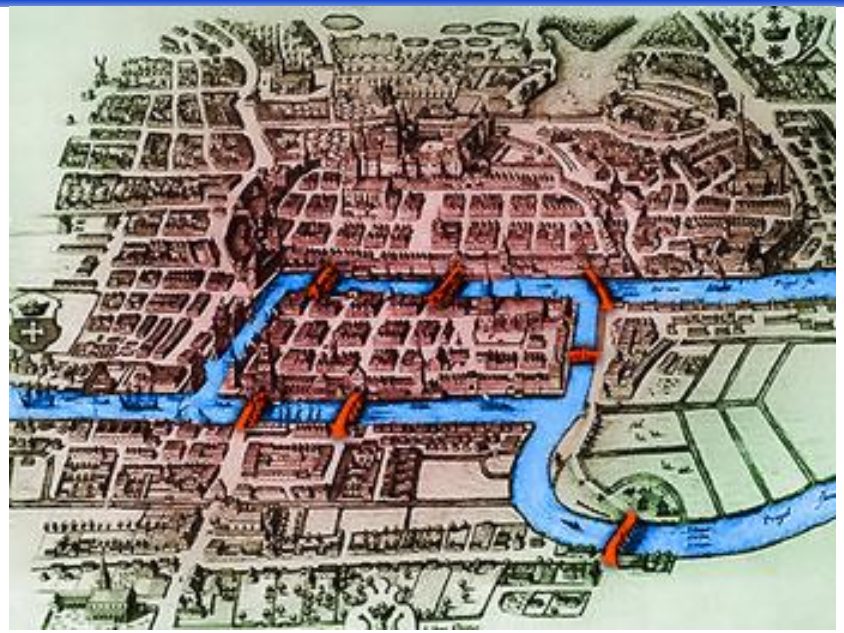
# Königsberg bridges

The subject of graph theory began in the year 1736 when the great mathematician Leonhard Euler published a paper giving the solution to the following puzzle:

The town of Königsberg in Prussia (now Kaliningrad in Russia) was built at a point where two branches of the Pregel River came together. It consisted of an island and some land along the river banks.

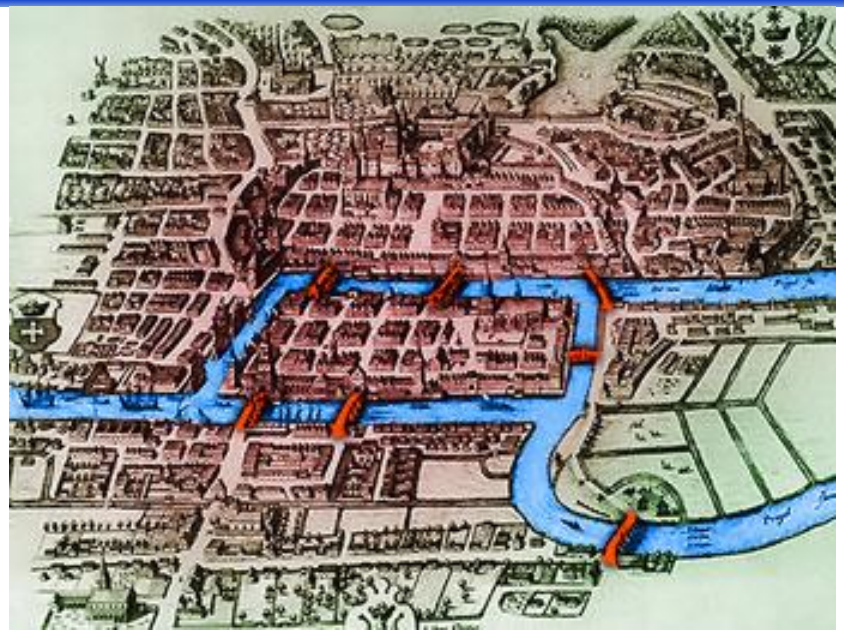
These were connected by 7 bridges.

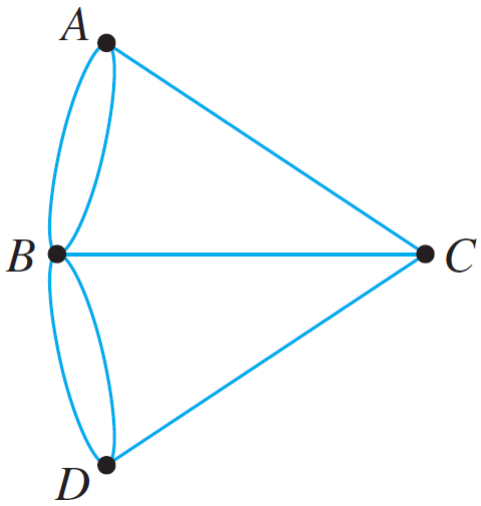
# Königsberg bridges



Question: Is it possible to take a walk around town, starting and ending at the same location and crossing each of the 7 bridges exactly once?

# Königsberg bridges





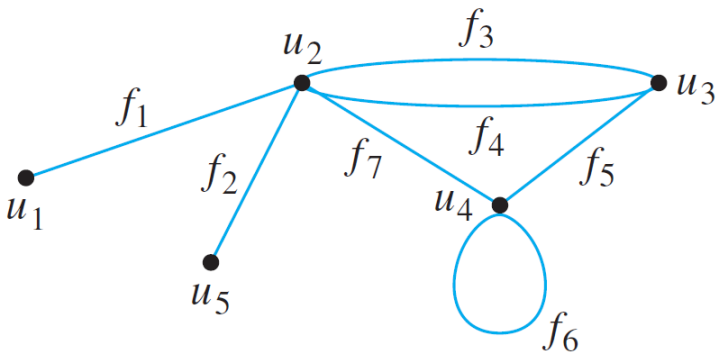

In terms of this graph, the question is:

Is it possible to find a route through the graph that starts and ends at some vertex, one of *A*, *B*, *C*, or *D*, and traverses each edge exactly once?

# Definitions

**Travel** in a graph is accomplished by moving from one vertex to another along a sequence of adjacent edges.

In the graph below, for instance, you can go from $u_1$ to $u_4$ by taking $f_1$ to $u_2$ and then $f_7$ to $u_4$. This is represented by writing $u_1 f_1 u_2 f_7 u_4$.




Or, you could take a roundabout route:

$$u_1 f_1 u_2 f_3 u_3 f_4 u_2 f_3 u_3 f_5 u_4 f_6 u_4 f_7 u_2 f_3 u_3 f_5 u_4.$$

# Walk, Trail, Path, Closed Walk, Circuit, Simple Circuit

## Definitions

Let $G$ be a graph, and let $v$ and $w$ be vertices of $G$.

A **walk from v to w** is a finite alternating sequence of adjacent vertices and edges of $G$. Thus a walk has the form

$$v_0\, e_1\, v_1\, e_2 \ldots v_{n-1}\, e_n\, v_n\,,$$

where the $v$'s represent vertices, the $e$'s represent edges, $v_0=v$, $v_n=w$, and for all $i \in \{1, 2, \ldots, n\}$, $v_{i-1}$ and $v_i$ are the endpoints of $e_i$.

The **trivial walk** from $v$ to $v$ consists of the single vertex $v$.

A **trail from v to w** is a walk from $v$ to $w$ that does not contain a repeated edge.

A **path from v to w** is a trail that does not contain a repeated vertex.

# Walk, Trail, Path, Closed Walk, Circuit, Simple Circuit

## Definitions

A **closed walk** is a walk that starts and ends at the same vertex.

A **circuit** (or **cycle**) is a closed walk that contains at least one edge and does not contain a repeated edge.

A **simple circuit** (or **simple cycle**) is a circuit that does not have any other repeated vertex except the first and last.

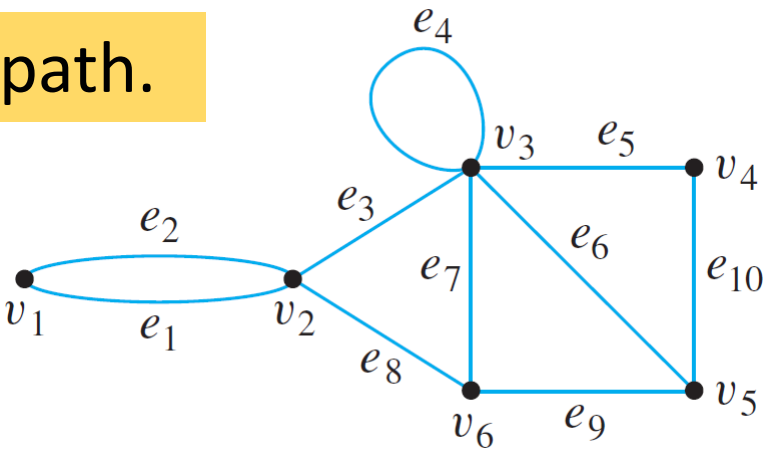
# Walk, Trail, Path, Closed Walk, Circuit, Simple Circuit

| | Repeated Edge? | Repeated Vertex? | Starts and Ends at Same Point? | Must Contain at Least One Edge? |
|---|---|---|---|---|
| **Walk** | allowed | allowed | allowed | no |
| **Trail** | no | allowed | allowed | no |
| **Path** | no | no | no | no |
| **Closed walk** | allowed | allowed | yes | no |
| **Circuit** | no | allowed | yes | yes |
| **Simple circuit** | no | first and last only | yes | yes |

Often a walk can be specified unambiguously by giving either a sequence of edges or a sequence of vertices.

# Walk, Trail, Path, Closed Walk, Circuit, Simple Circuit

In this graph, determine which of the following walks are trails, paths, circuits, or simple circuits.




a. $v_1 e_1 v_2 e_3 v_3 e_4 v_3 e_5 v_4$ Trail; not a path.

b. $e_1 e_3 e_5 e_5 e_6$ Walk; not a trail.

c. $v_2 v_3 v_4 v_5 v_3 v_6 v_2$ Circuit; not a simple circuit.

d. $v_2 v_3 v_4 v_5 v_6 v_2$ Simple circuit.

e. $v_1 e_1 v_2 e_1 v_1$ Closed walk; not a circuit.

f. $v_1$ Closed walk; not a circuit.

# Notes

Because most of the major developments in graph theory have happened relatively recently and in a variety of different contexts, the terms used in the subject have not been standardized.

| Susanna Epp's book | Others |
|---|---|
| Graph | Multigraph |
| Simple graph | Graph |
| Vertex | Node |
| Edge | Arc |
| Trail | Path |
| Path | Simple path |
| Simple circuit | Cycle |

The terminology in this book is among the most common, but if you consult other sources, be sure to check their definitions.

# Connectedness

A graph is connected if it is possible to travel from any vertex to any other vertex along a sequence of adjacent edges of the graph.

## Definition: Connectedness

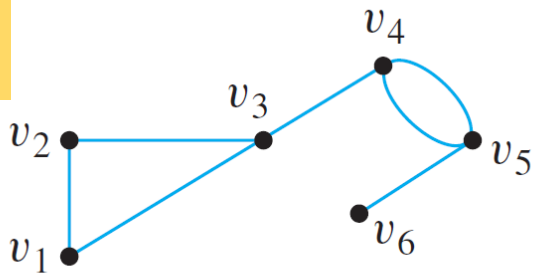
**Two vertices** $v$ and $w$ of a graph $G$ are **connected** if, and only if, there is a walk from $v$ to $w$.

**The graph G is connected** if, and only if, given *any* two vertices $v$ and $w$ in $G$, there is a walk from $v$ to $w$. Symbolically,

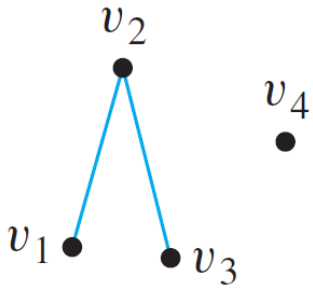
$G$ is connected iff $\forall$ vertices $v, w \in V(G)$, $\exists$ a walk from $v$ to $w$.

# Example: Which of the following graphs are connected?

Yes



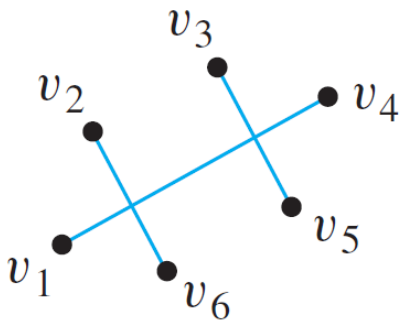

(a)





(b)

No




(c)

No

Some useful facts relating circuits and connectedness are collected in the following lemma.

## Lemma 10.2.1

Let $G$ be a graph.

a. If $G$ is connected, then any two distinct vertices of $G$ can be connected by a path.

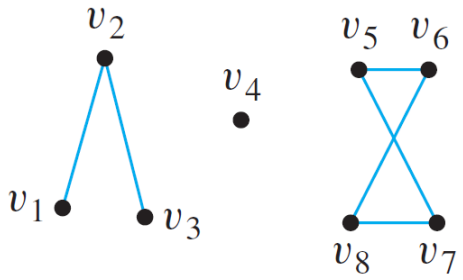
b. If vertices $v$ and $w$ are part of a circuit in $G$ and one edge is removed from the circuit, then there still exists a trail from $v$ to $w$ in $G$.

c. If $G$ is connected and $G$ contains a circuit, then an edge of the circuit can be removed without disconnecting $G$.

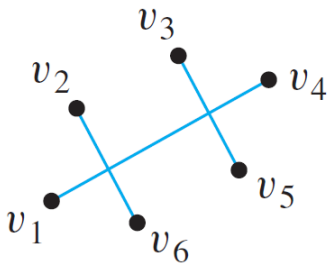
# Connected Component

The graphs in (*b*) and (*c*) are both made up of three pieces, each of which is itself a connected graph.

A *connected component* of a graph is a connected subgraph of largest possible size.




(b)




(c)

## Definition: Connected Component

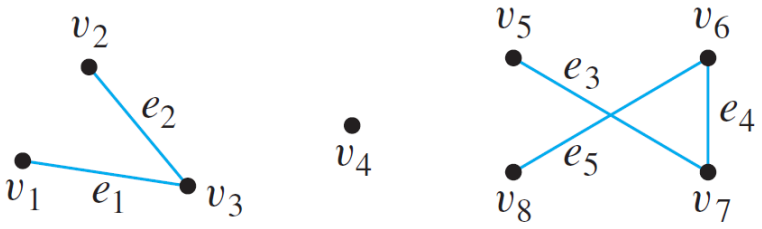
A graph *H* is a **connected component** of a graph *G* if, and only if,

1. The graph *H* is a subgraph of *G*;
2. The graph *H* is connected; and
3. No connected subgraph of *G* has *H* as a subgraph and contains vertices or edges that are not in *H*.

The fact is that any graph is a kind of union of its connected components.

## Connected Component

Find all connected components of the following graph *G*.




*G* has 3 connected components $H_1$, $H_2$ and $H_3$ with vertex sets $V_1$, $V_2$ and $V_3$ and edge sets $E_1$, $E_2$ and $E_3$ , where

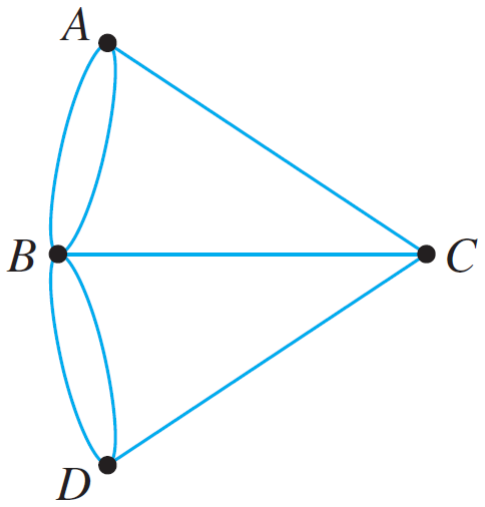
$$V_1 = \{v_1, v_2, v_3\}, \quad E_1 = \{e_1, e_2\}$$

$$V_2 = \{v_4\}, \quad E_2 = \emptyset$$

$$V_3 = \{v_5, v_6, v_7, v_8\}, \quad E_3 = \{e_3, e_4, e_5\}$$

# Euler Circuits

Now, let's go back to the puzzle of the Königsberg bridges.




Is it possible to find a route through the graph that starts and ends at some vertex, one of *A*, *B*, *C*, or *D*, and traverses each edge exactly once?

## Definition: Euler Circuit

Let *G* be a graph. An **Euler circuit** for *G* is a circuit that contains every vertex and every edge of *G*.

That is, an Euler circuit for *G* is a sequence of adjacent vertices and edges in *G* that has at least one edge, starts and ends at the same vertex, uses every vertex of *G* at least once, and uses every edge of *G* exactly once.

## Definition: Eulerian Graph

An **Eulerian graph** is a graph that contains an Euler circuit.

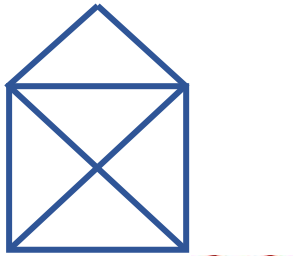
## Theorem 10.2.2

If a graph has an Euler circuit, then every vertex of the graph has positive even degree.

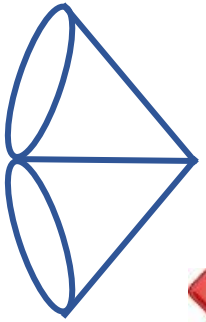
## Contrapositive Version of Theorem 10.2.2

If some vertex of a graph has odd degree, then the graph does not have an Euler circuit.

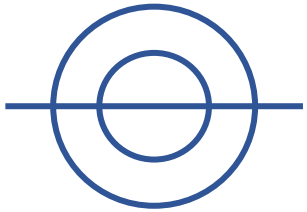
# Does each of the following graphs have an Euler circuit?



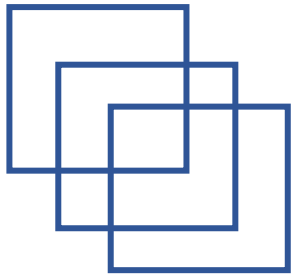
(1) ✗



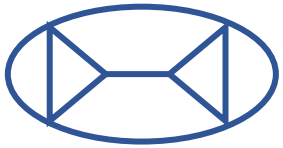
(2) ✗



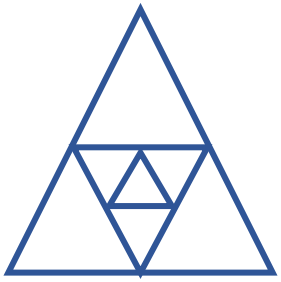
(3) ✗



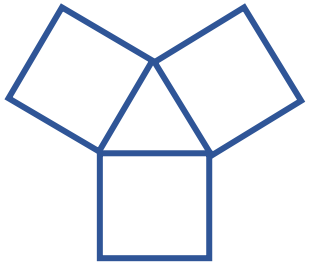
(4) ✓



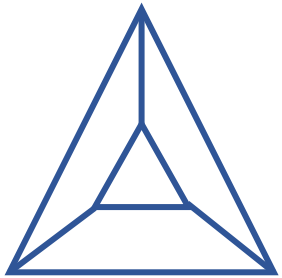
(5) ✗



(6) ✓



(7) ✓

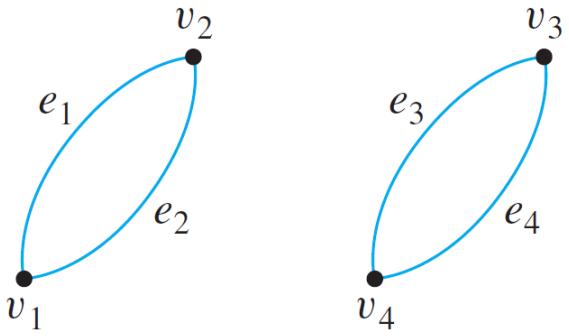


(8) ✗

# Is the converse of Theorem 10.2.2 true?

If every vertex of a graph has even degree, then the graph has an Euler circuit.

Not true!



## Theorem 10.2.3

If a graph *G* is connected and the degree of every vertex of *G* is a positive even integer, then *G* has an Euler circuit.

The proof of Theorem 10.2.3 is constructive: It contains an algorithm to find an Euler circuit for any connected graph in which every vertex has even degree.

## Theorem 10.2.4

A graph *G* has an Euler circuit if, and only if, *G* is connected and every vertex of *G* has positive even degree.

A corollary to Theorem 10.2.4 gives a criterion for determining when it is possible to find a walk from one vertex of a graph to another, passing through every vertex of the graph at least once and every edge of the graph exactly once.

## Definition: Euler Trail

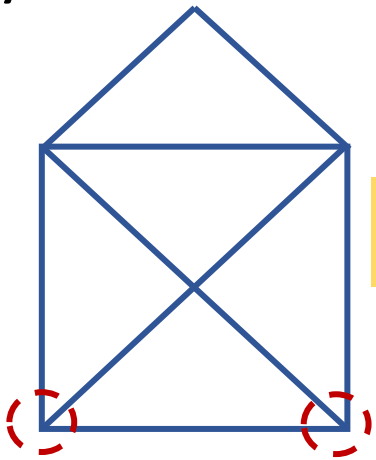
Let *G* be a graph, and let *v* and *w* be two distinct vertices of *G*. An **Euler trail/path from *v* to *w*** is a sequence of adjacent edges and vertices that starts at *v*, ends at *w*, passes through every vertex of *G* at least once, and traverses every edge of *G* exactly once.

## Corollary 10.2.5

Let *G* be a graph, and let *v* and *w* be two distinct vertices of *G*. There is an Euler trail from *v* to *w* if, and only if, *G* is connected, *v* and *w* have odd degree, and all other vertices of *G* have positive even degree.

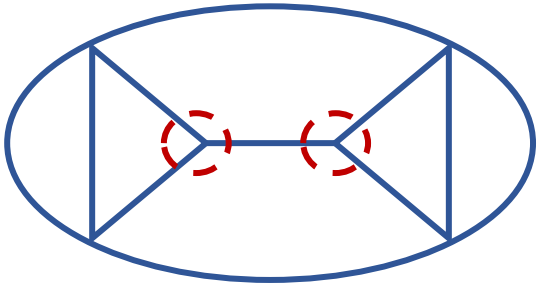
## Euler Circuits

The following graphs do not have an Euler circuit. Do they have an Euler trail?



Yes

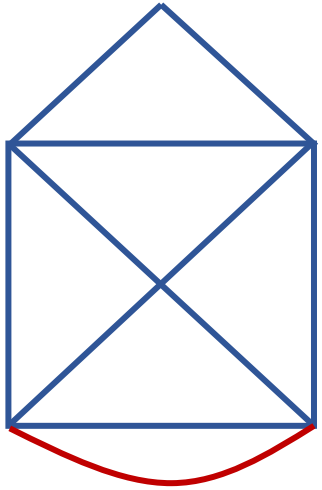
(1)

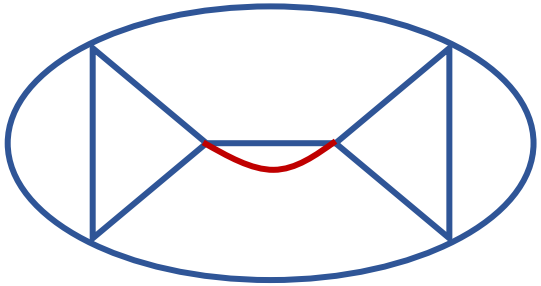


Yes

(5)

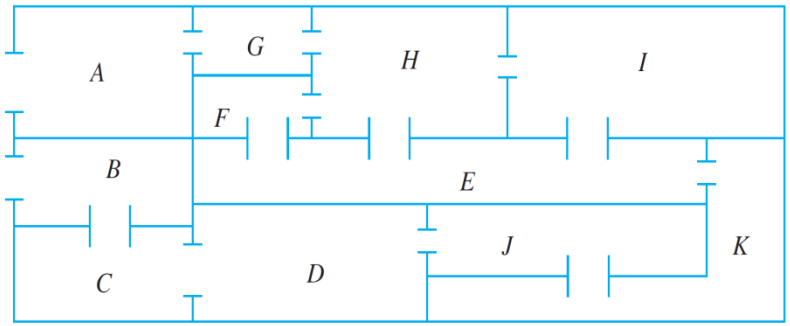
Adding an edge between the two vertices with odd degree will give us an Euler circuit.

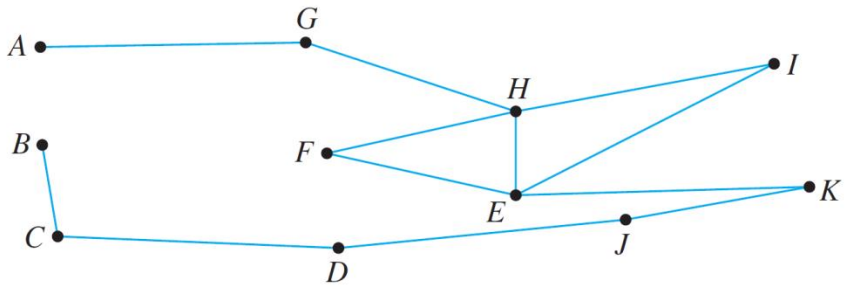




## Euler Circuits

The floor plan shown below is for a house that is open for public viewing. Is it possible to find a trail that starts in room *A*, ends in room *B*, and passes through every interior doorway of the house exactly once? If so, find such a trail.




Each vertex of this graph has even degree except for *A* and *B*, each of which has degree 1.

Hence by Corollary 10.2.5, there is an Euler path from *A* to *B*. One such trail is *AGHFEIHEKJDCB.*

# Hamiltonian Circuits

Recall Theorem 10.2.4:

**Theorem 10.2.4**

A graph *G* has an Euler circuit if, and only if, *G* is connected and every vertex of *G* has positive even degree.

A related question:

Given a graph *G*, is it possible to find a circuit for *G* in which all the *vertices* of *G* (except the first and the last) appear exactly once?

In 1859 the Irish mathematician Sir William Rowan Hamilton introduced a puzzle in the shape of a dodecahedron (DOH-dek-a-HEE-dron). (Figure 10.2.6 contains a drawing of a dodecahedron, which is a solid figure with 12 identical pentagonal faces.)

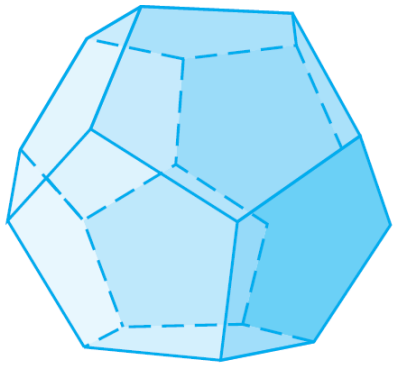
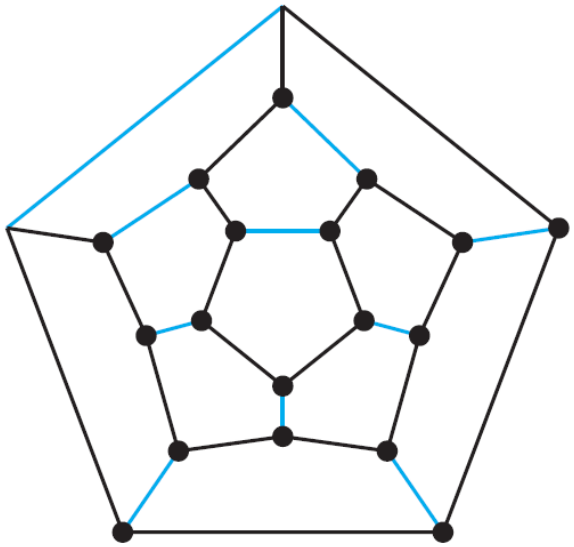


**Figure 10.2.6** Dodecahedron

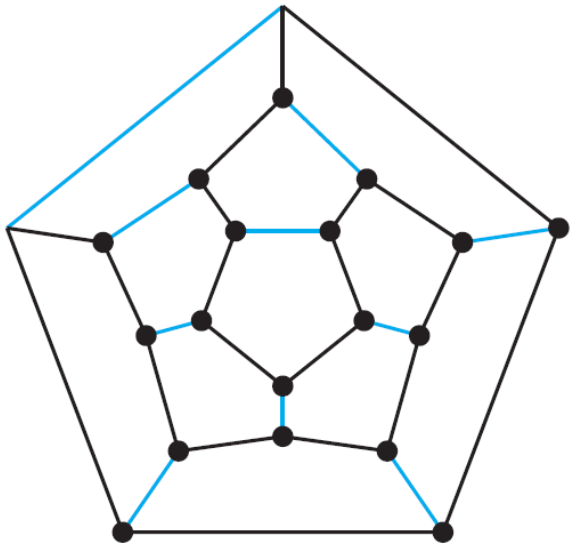
Each vertex was labeled with the name of a city — London, Paris, Hong Kong, New York, and so on.

The problem Hamilton posed was to start at one city and tour the world by visiting each other city exactly once and returning to the starting city.

One way to solve the puzzle is to imagine the surface of the dodecahedron stretched out and laid flat in the plane, as follows:



The circuit denoted with black lines is one solution. Note that although every city is visited, many edges are omitted from the circuit. (More difficult versions of the puzzle required that certain cities be visited in a certain order.)



## Definition: Hamiltonian Circuit

Given a graph *G*, a **Hamiltonian circuit** for *G* is a simple circuit that includes every vertex of *G*.

That is, a Hamiltonian circuit for *G* is a sequence of adjacent vertices and distinct edges in which every vertex of *G* appears exactly once, except for the first and the last, which are the same.

## Definition: Hamiltonian Graph

A **Hamiltonian graph** (also called **Hamilton graph**) is a graph that contains a Hamiltonian circuit.

Note that although an Euler circuit for a graph *G* must include every vertex of *G*, it may visit some vertices more than once and hence may not be a Hamiltonian circuit.

On the other hand, a Hamiltonian circuit for *G* does not need to include all the edges of *G* and hence may not be an Euler circuit.

Despite the analogous-sounding definitions of Euler and Hamiltonian circuits, the mathematics of the two are very different.

Theorem 10.2.4 gives a simple criterion for determining whether a given graph has an Euler circuit.

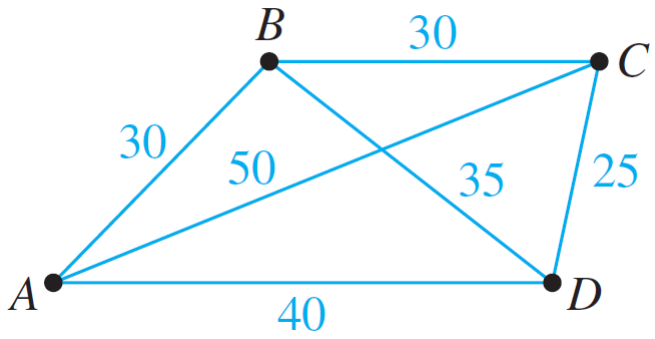
Unfortunately, there is no analogous criterion for determining whether a given graph has a Hamiltonian circuit, nor is there even an efficient algorithm for finding such a circuit.

**Theorem 10.2.4**

A graph *G* has an Euler circuit if, and only if, *G* is connected and every vertex of *G* has positive even degree.

# Travelling Salesman Problem

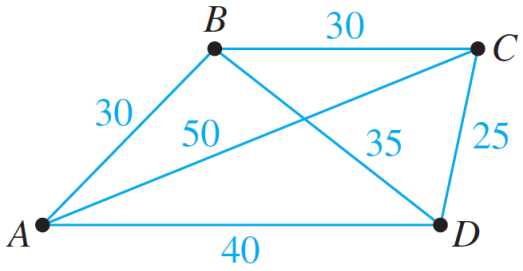
Imagine that the drawing below is a map showing four cities and the distances in kilometers between them.




Suppose that a salesman must travel to each city exactly once, starting and ending in city *A*. Which route from city to city will minimize the total distance that must be travelled?

## Travelling Salesman Problem

This problem can be solved by writing all possible Hamiltonian circuits starting and ending at *A* and calculating the total distance travelled for each.




| Route | Total Distance (In Kilometers) | |
|---|---|---|
| $ABCDA$ | $30 + 30 + 25 + 40 = 125$ | |
| $ABDCA$ | $30 + 35 + 25 + 50 = 140$ | |
| $ACBDA$ | $50 + 30 + 35 + 40 = 155$ | |
| $ACDBA$ | 140 | [$ABDCA$ backwards] |
| $ADBCA$ | 155 | [$ACBDA$ backwards] |
| $ADCBA$ | 125 | [$ABCDA$ backwards] |

Thus either route *ABCDA* or *ADCBA* gives a minimum total distance of 125 km.

## Travelling Salesman Problem

The general travelling salesman problem involves finding a Hamiltonian circuit to minimize the total distance travelled for an arbitrary graph with *n* vertices in which each edge is marked with a distance.

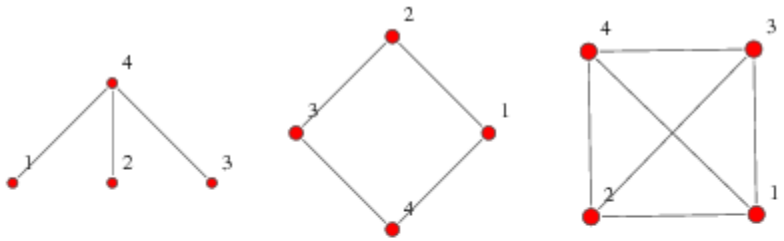
One way to solve the general problem is to use the previous method: Write down all Hamiltonian circuits starting and ending at a particular vertex, compute the total distance for each, and pick one for which this total is minimal.

However, this is impractical for even medium-sized values of *n*. For $n = 30$ vertices, there would be $(29!)/2 \approx 4.42 \times 10^{30}$ Hamiltonian circuits starting and ending at a particular vertex to check. If each circuit could be found and its total distance computed in just one nanosecond, it would take approximately $1.4 \times 10^{14}$ years to compute!

At present, there is no known algorithm for solving the general travelling salesman problem that is more efficient.

However, there are efficient algorithms that find "pretty good" solutions — that is, circuits that, while not necessarily having the least possible total distances, have smaller total distances than most other Hamiltonian circuits.

# 10.3 Matrix Representations of Graphs




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

# Matrices

## Definition: Matrix

An $m \times n$ (read "$m$ by $n$") **matrix** A over a set $S$ is a rectangular array of elements of $S$ arranged into $m$ rows and $n$ columns.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{bmatrix}$$

← $i$th row of **A**

$j$th column of **A**

We write $\mathbf{A} = (a_{ij})$.

## Matrices

If **A** and **B** are matrices, then **A** = **B** if, and only if, **A** and **B** have the same size and the corresponding entries of **A** and **B** are all equal; that is,

$$a_{ij} = b_{ij} \text{ for all } i = 1, 2, \cdots, m \text{ and } j = 1, 2, \cdots, n.$$

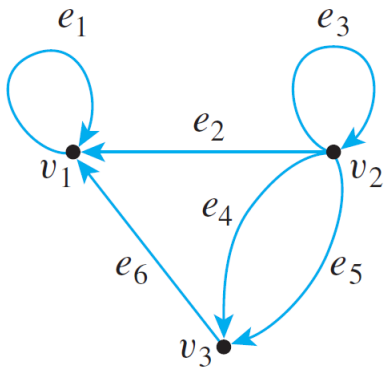
A matrix for which the numbers of rows and columns are equal is called a **square matrix**.

If **A** is a square matrix of size $n \times n$, then the **main diagonal** of **A** consists of all the entries $a_{11}, a_{22}, \cdots a_{nn}$.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1i} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2i} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i1} & a_{i2} & \dots & a_{ii} & \dots & a_{in} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{ni} & \dots & a_{nn} \end{bmatrix}$$

main diagonal of **A**

# Matrices and Directed Graphs




Directed Graph $G$

(a)

$$\mathbf{A} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix

(b)

Figure 10.3.1 A Directed Graph and Its Adjacency Matrix

This graph $G$ is represented by the matrix $\mathbf{A} = (a_{ij})$ for which $a_{ij}$ = number of arrows from $v_i$ to $v_j$ for all $i$ = 1, 2, 3 and $j$ = 1, 2, 3.

**A** is called the **adjacency matrix** of $G$.

Another common representation of a graph is the adjacency list, which is covered in algorithms module.

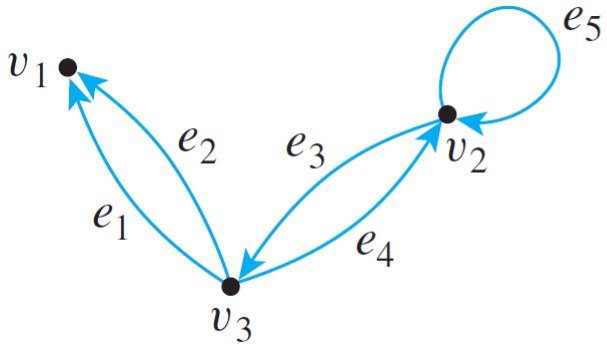
## Definition: Adjacency Matrix of a Directed Graph

Let $G$ be a directed graph with ordered vertices $v_1, v_2, \ldots v_n$. The **adjacency matrix of G** is the $n \times n$ matrix $\mathbf{A} = (a_{ij})$ over the set of non-negative integers such that

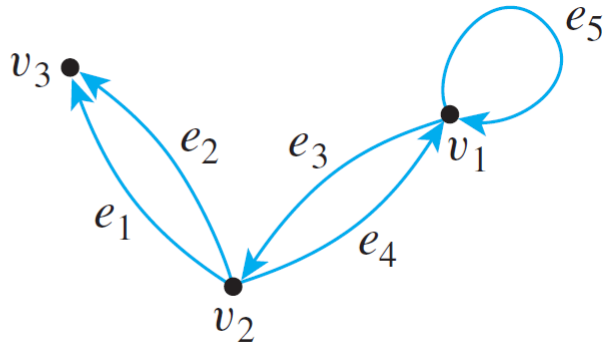
$$a_{ij} = \text{the number of arrows from } v_i \text{ to } v_j$$

for all $i, j = 1, 2, \ldots, n$.

Example: Find the adjacency matrices of the two directed graphs below.




(a)




(b)

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 2 & 1 & 0 \end{bmatrix} \end{array}$$

(a)

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b)

# Matrices and Undirected Graphs

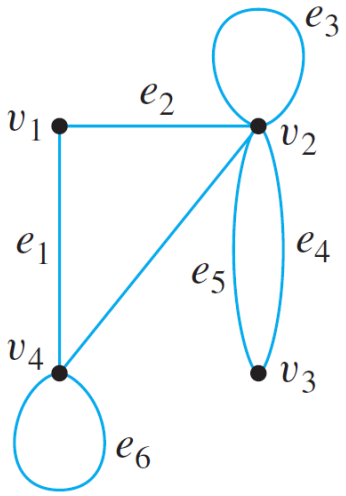
## Definition: Adjacency Matrix of an Undirected Graph

Let $G$ be an undirected graph with ordered vertices $v_1, v_2, \ldots v_n$. The **adjacency matrix of G** is the $n \times n$ matrix $\mathbf{A} = (a_{ij})$ over the set of non-negative integers such that

$$a_{ij} = \text{the number of edges connecting } v_i \text{ and } v_j$$

for all $i, j = 1, 2, \ldots, n$.

Example: Find the adjacency matrix for the graph *G* shown below.




$$\mathbf{A} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

Note that the matrix is symmetric.

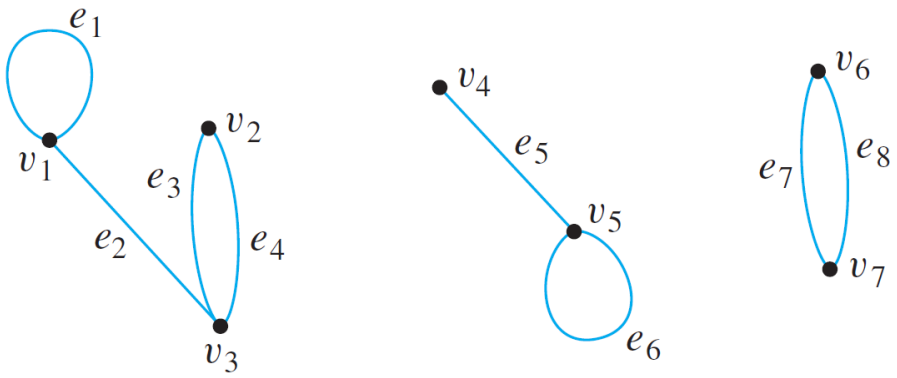
## Definition: Symmetric Matrix

An $n \times n$ square matrix A = $(a_{ij})$ is called **symmetric** if, and only if, for all $i, j = 1, 2, \ldots, n$,

$$a_{ij} = a_{ji}.$$

# Matrices and Connected Components

Consider a graph *G*, as shown below, that consists of several connected components.

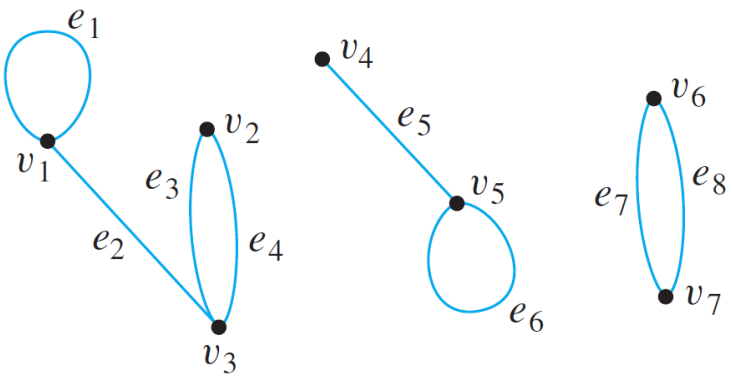



Adjacency matrix of *G*:

$$\mathbf{A} = \left[\begin{array}{ccc:c:cc:cc} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \end{array}\right]$$

As you can see, **A** consists of square matrix blocks (of different sizes) down its diagonal and blocks of 0's everywhere else.

The reason is that vertices in each connected component share no edges with vertices in other connected components.




For instance, since $v_1$, $v_2$, and $v_3$ share no edges with $v_4$, $v_5$, $v_6$, or $v_7$, all entries in the top three rows to the right of the third column are 0 and all entries in the left three columns below the third row are also 0.

$$\mathbf{A} = \left[\begin{array}{ccc:ccc:cc} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \end{array}\right]$$

Sometimes matrices whose entries are all 0's are themselves denoted 0. If this convention is followed here, **A** is written as:

$$\mathbf{A} = \left[\begin{array}{ccc|c|c} 1 & 0 & 1 & & \\ 0 & 0 & 2 & & \\ 1 & 2 & 0 & & \\ \hline & & & \begin{matrix} 0 & 1 \\ 1 & 1 \end{matrix} & \\ \hline & & & & \begin{matrix} 0 & 2 \\ 2 & 0 \end{matrix} \end{array}\right]$$

The previous reasoning can be generalized to prove the following theorem:

## Theorem 10.3.1

Let $G$ be a graph with connected components $G_1$, $G_1$, …, $G_k$. If there are $n_i$ vertices in each connected component $G_i$ and these vertices are numbered consecutively, then the adjacency matrix of $G$ has the form:

$$\begin{bmatrix} A_1 & O & O & & O & O \\ O & A_2 & O & \cdots & O & O \\ O & O & A_3 & & O & O \\ & \vdots & & & \vdots & \\ O & O & O & \cdots & O & A_k \end{bmatrix}$$

where each $A_i$ is $n_i \times n_i$ adjacency matrix of $G_i$, for all $i = 1, 2, \ldots, k$, and the O's represent matrices whose entries are all 0s.

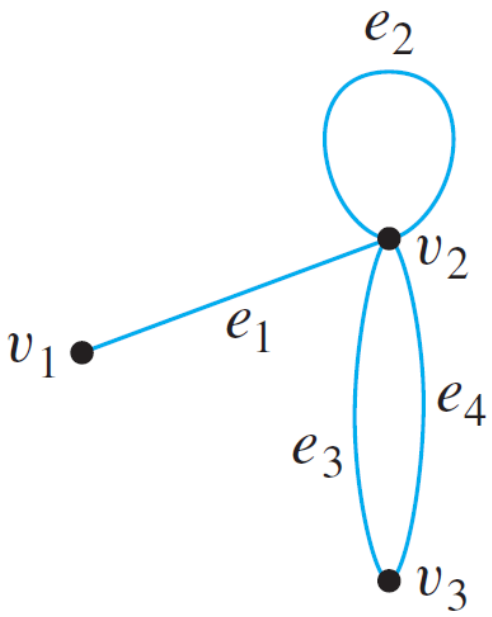
# Counting Walks of Length *N*

A **walk** in a graph consists of an alternating sequence of vertices and edges.

If repeated edges are counted each time they occur, then the number of edges in the sequence is called the **length** of the walk.

For instance, the walk $v_2e_3v_3e_4v_2e_2v_2e_3v_3$ has length 4 (counting $e_3$ twice).

Example: Consider the following graph *G*.
How many distinct walks of length 2 connect $v_2$ and $v_2$?




One walk of length 2 from $v_2$ to $v_2$ via $v_1$:
$v_2e_1v_1e_1v_2$.

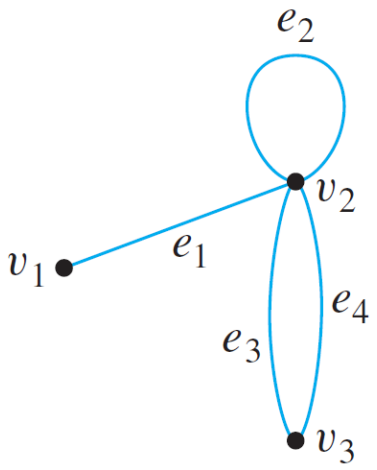
One walk of length 2 from $v_2$ to $v_2$ via $v_2$:
$v_2e_2v_2e_2v_2$.

Four walks of length 2 from $v_2$ to $v_2$ via $v_3$:
$v_2e_3v_3e_4v_2$ ,
$v_2e_4v_3e_3v_2$,
$v_2e_3v_3e_3v_2$,
$v_2e_4v_3e_4v_2$.

Total = 6

The general question of finding the number of walks that have a given length and connect two particular vertices of a graph can easily be answered using matrix multiplication.

Consider the adjacency matrix **A** of the graph *G*.

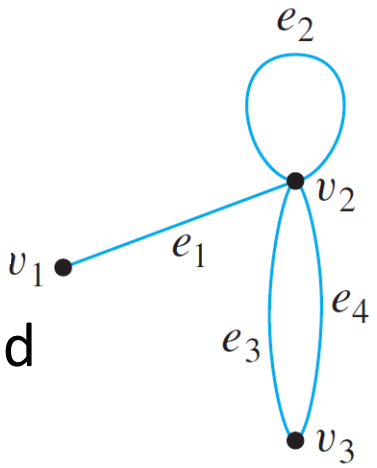



$$\mathbf{A} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 2 \\ 0 & 2 & 0 \end{bmatrix} \end{array}.$$

## Counting Walks of Length *N*

Compute $\mathbf{A}^2$:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 2 \\ 0 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 2 \\ 0 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 6 & 2 \\ 2 & 2 & 4 \end{bmatrix}.$$




Note that the entry in the second row and the second column is 6, which equals the number of walks of length 2 from $v_2$ to $v_2$.

Reason: To compute $a_{22}$, you multiply the second row of **A** times the second column of **A** to obtain a sum of three terms:

$$\begin{bmatrix} 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = 1 \cdot 1 + 1 \cdot 1 + 2 \cdot 2.$$

More generally, if **A** is the adjacency matrix of a graph *G*, the *i j*-th entry of $\mathbf{A}^2$ equals the number of walks of length 2 connecting the *i*-th vertex to the *j*-th vertex of *G*.

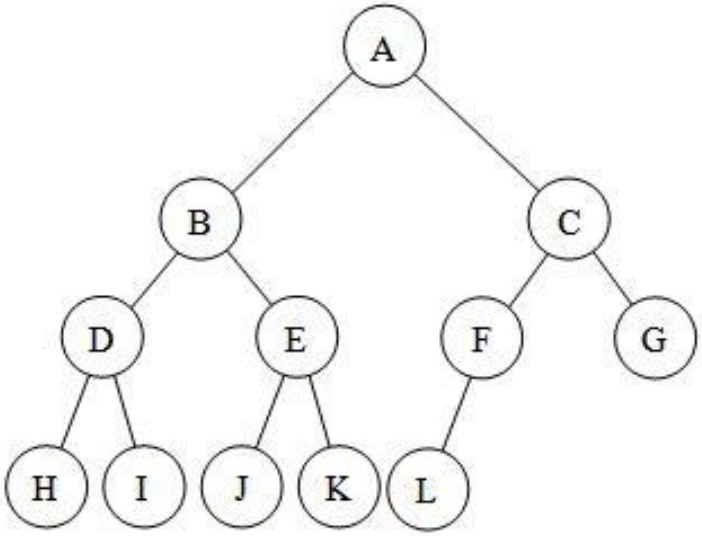
Even more generally, if *n* is any positive integer, the *i j*-th entry of $\mathbf{A}^n$ equals the number of walks of length *n* connecting the *i*-th and the *j*-th vertices of *G*.

## Theorem 10.3.2

If *G* is a graph with vertices $v_1, v_2, \ldots, v_m$ and **A** is the adjacency matrix of *G*, then for each positive integer *n* and for all integers $i, j = 1, 2, \ldots, m$,
the *ij*-th entry of $\mathbf{A}^n$ = the number of walks of length *n* from $v_i$ to $v_j$.

# 12. Graphs and Trees 2

# 10.5 Trees



# Definition

## Definition: Tree

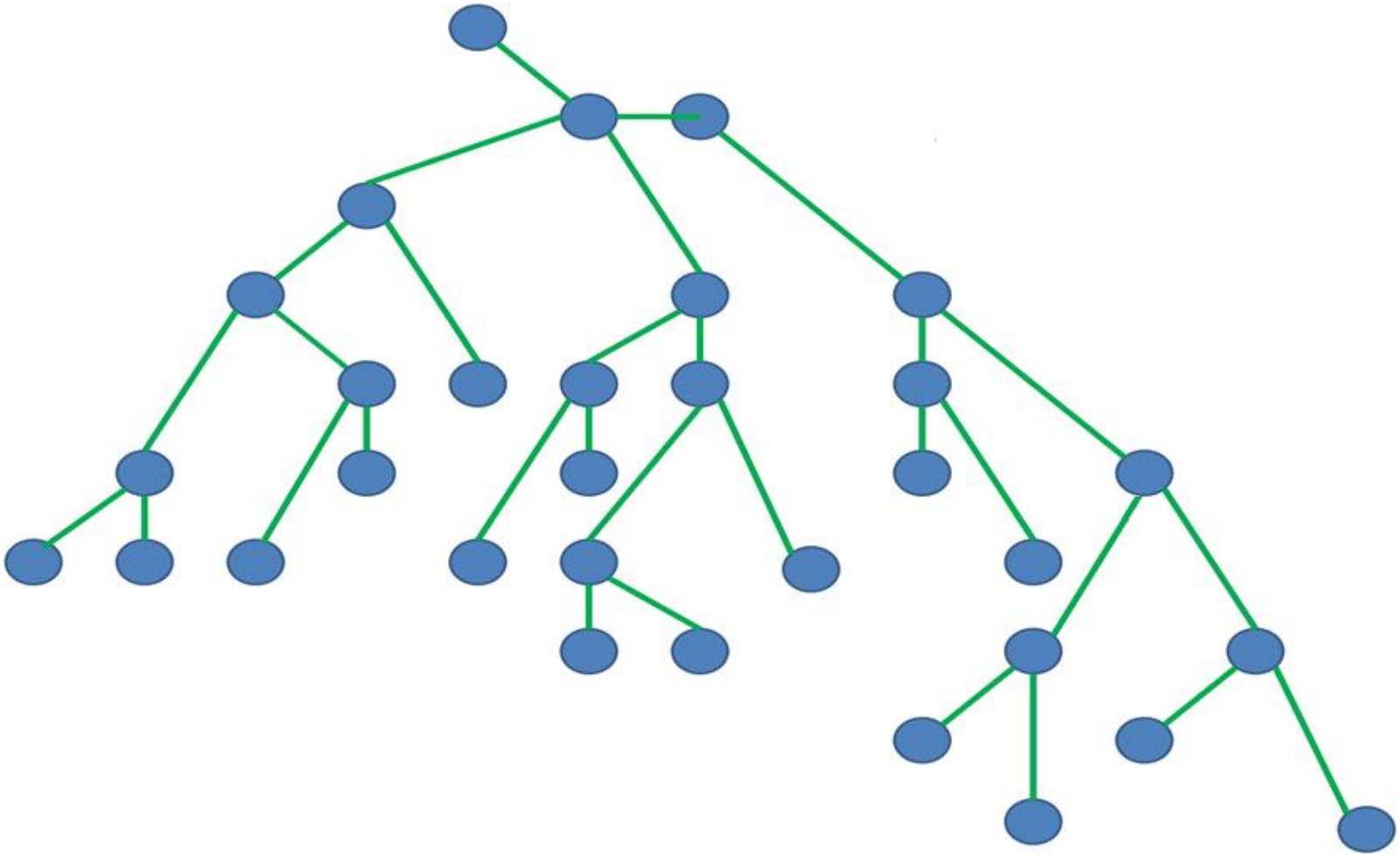
A **graph** is said to be **circuit-free** if, and only if, it has no circuits.

A graph is called a **tree** if, and only if, it is circuit-free and connected.

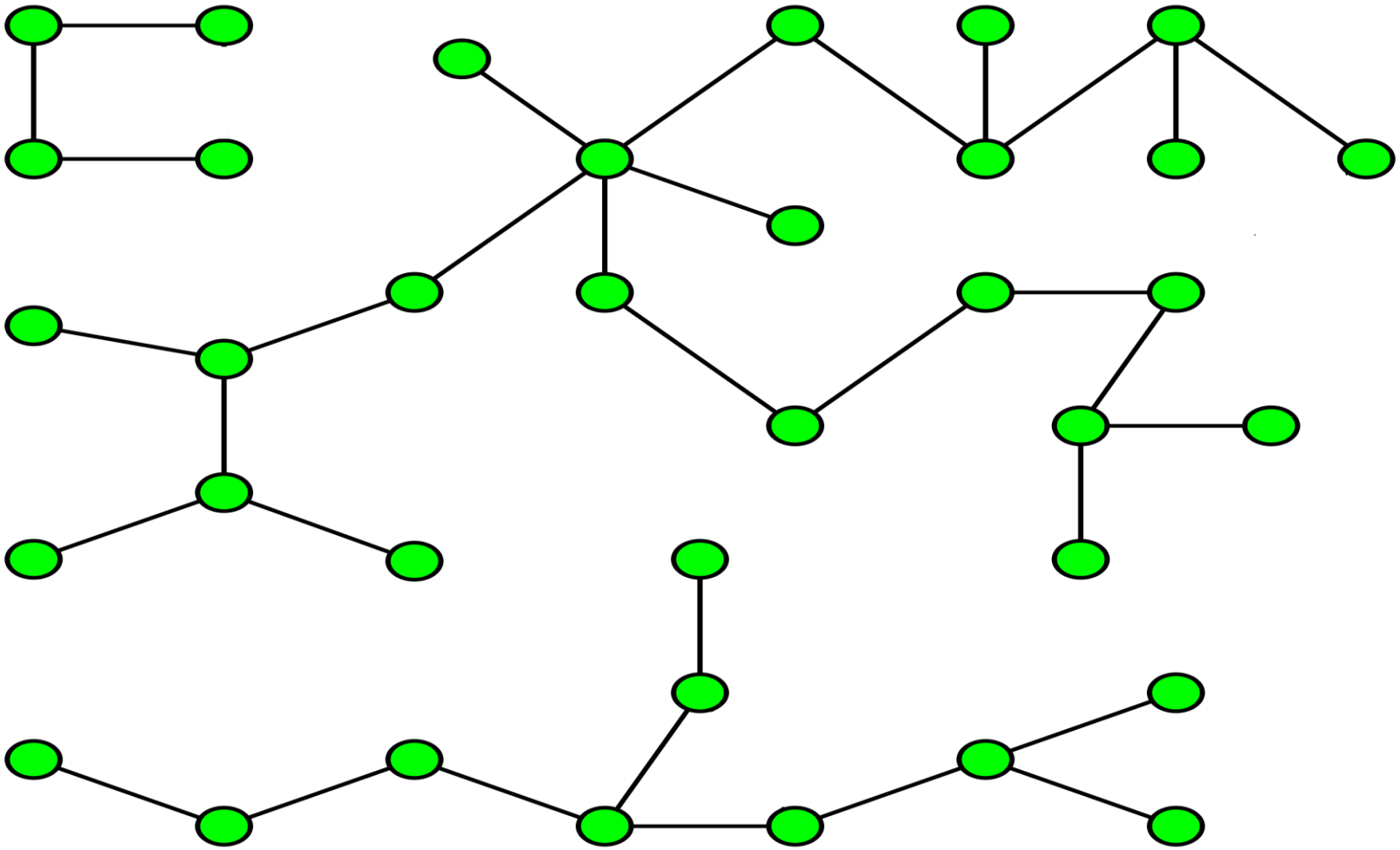
A **trivial tree** is a graph that consists of a single vertex.

A graph is called a **forest** if, and only if, it is circuit-free and not connected.

# Tree



# Forest

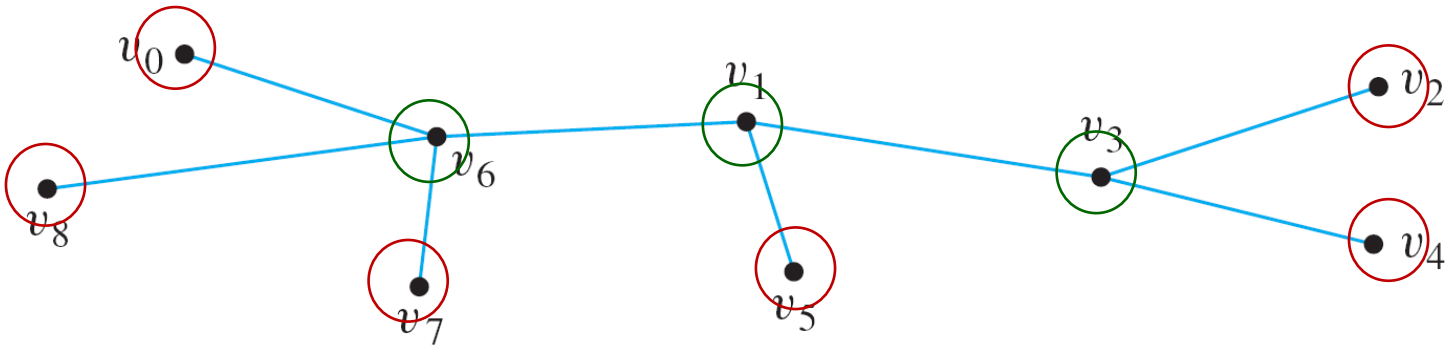


## Definitions: Terminal vertex (leaf) and internal vertex

Let $T$ be a tree. If $T$ has only one or two vertices, then each is called a **terminal vertex** (or **leaf**).

If $T$ has at least three vertices, then a vertex of degree 1 in $T$ is called a **terminal vertex** (or **leaf**), and a vertex of degree greater than 1 in $T$ is called an **internal vertex**.

Example: Find all terminal vertices and all internal vertices in the following tree:




Terminal vertices: $v_0$, $v_2$, $v_4$, $v_5$, $v_7$ and $v_8$.

Internal vertices: $v_6$, $v_1$ and $v_3$.

## Theorem 10.5.2

Any tree with $n$ vertices ($n > 0$) has $n - 1$ edges.

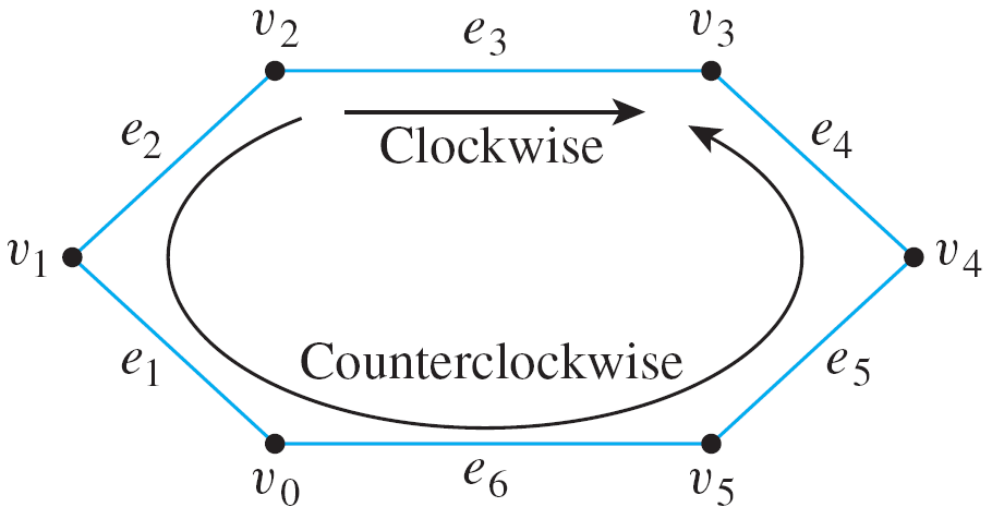
## Lemma 10.5.3

If $G$ is any connected graph, $C$ is any circuit in $G$, and one of the edges of $C$ is removed from $G$, then the graph that remains is still connected.

Essentially, the reason why Lemma 10.5.3 is true is that any two vertices in a circuit are connected by two distinct paths.

It is possible to draw the graph so that one of these goes "clockwise" and the other goes "counter-clockwise" around the circuit.

For example, in the circuit shown below:




The clockwise path from $v_2$ to $v_3$ is

$$v_2\ e_3\ v_3$$

and the counter-clockwise path from $v_2$ to $v_3$ is

$$v_2\ e_2\ v_1\ e_1\ v_0\ e_6\ v_5\ e_5\ v_4\ e_4\ v_3$$

# 10.6 Rooted Trees

A rooted tree is a tree in which one vertex has been distinguished from the others and is designated the *root*.

## Definitions: Rooted Tree, Level, Height

A **rooted tree** is a tree in which there is one vertex that is distinguished from the others and is called the **root**.

The **level** of a vertex is the number of edges along the unique path between it and the root.

The **height** of a rooted tree is the maximum level of any vertex of the tree.

## Definitions: Child, Parent, Sibling, Ancestor, Descendant

Given the root or any internal vertex *v* of a rooted tree, the **children** of *v* are all those vertices that are adjacent to *v* and are one level farther away from the root than *v*.

If *w* is a child of *v*, then *v* is called the **parent** of *w*, and two distinct vertices that are both children of the same parent are called **siblings**.

Given two distinct vertices *v* and *w*, if *v* lies on the unique path between *w* and the root, then *v* is an **ancestor** of *w*, and *w* is a **descendant** of *v*.

## Definitions

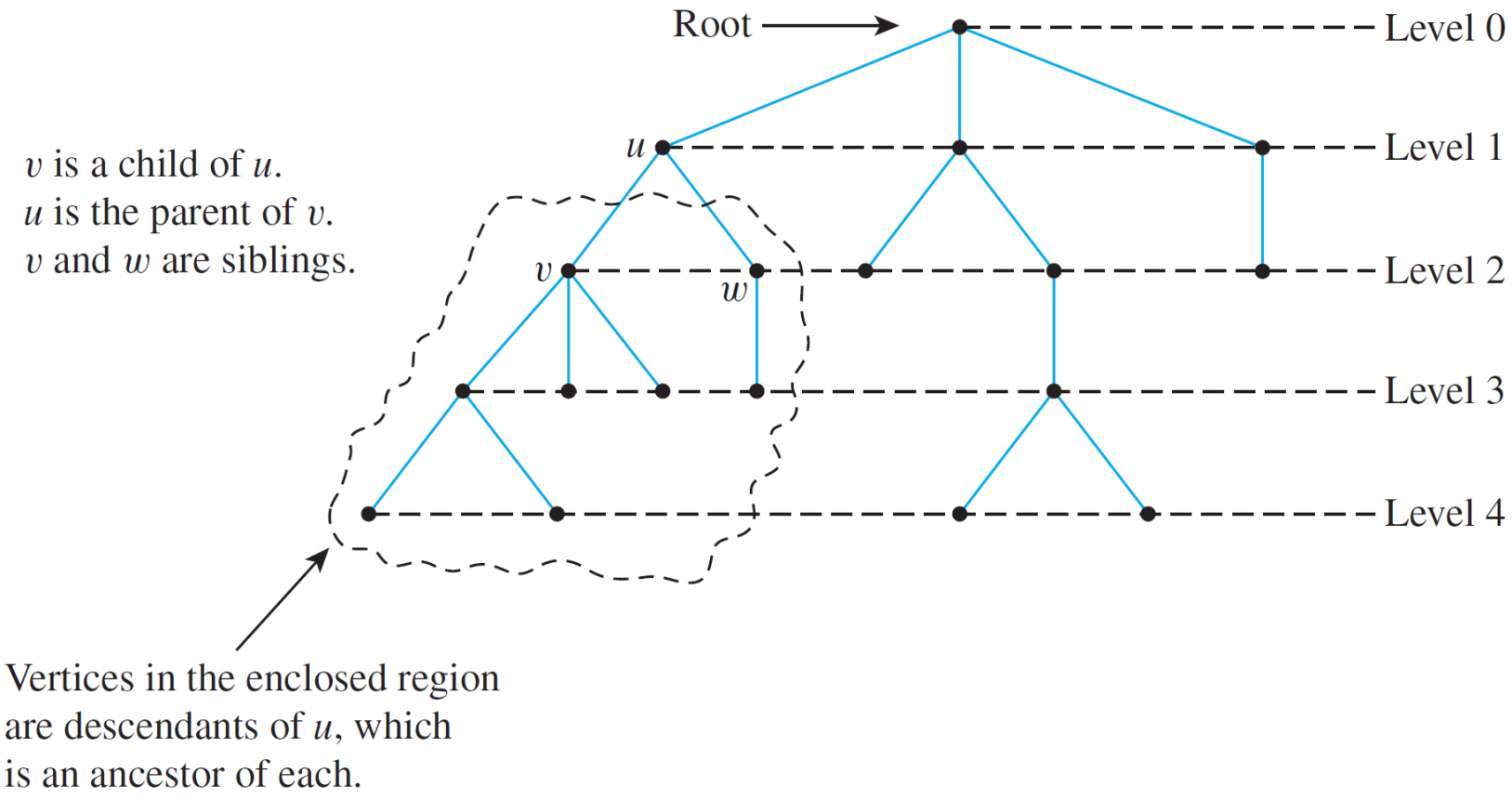
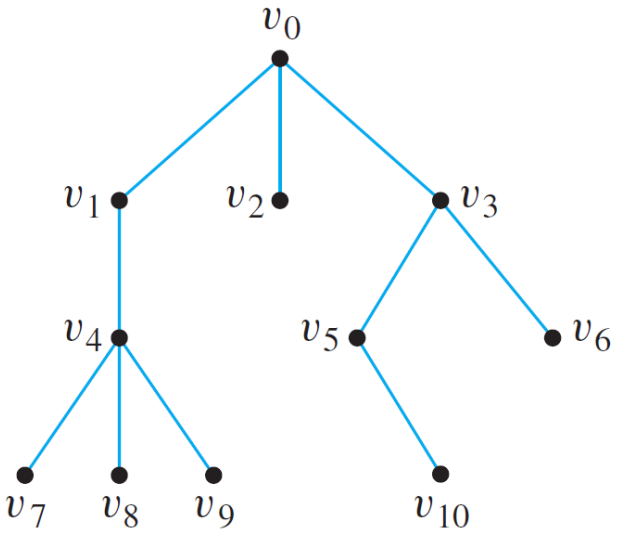



**Figure 10.6.1** A Rooted Tree

# Example: Consider the tree with root $v_0$ shown below.




a. What is the level of $v_5$? 2

b. What is the level of $v_0$? 0

c. What is the height of this rooted tree? 3

d. What are the children of $v_3$? $v_5$ and $v_6$

e. What is the parent of $v_2$? $v_0$

f. What are the siblings of $v_8$? $v_7$ and $v_9$

g. What are the descendants of $v_3$? $v_5$, $v_6$ and $v_{10}$

# Binary Trees

## Definitions: Binary Tree, Full Binary Tree

A **binary tree** is a rooted tree in which every parent has at most two children. Each child is designated either a **left child** or a **right child** (but not both), and every parent has at most one left child and one right child.

A **full binary tree** is a binary tree in which each parent has exactly two children.

# Binary Trees

## Definitions: Left Subtree, Right Subtree

Given any parent *v* in a binary tree *T*, if *v* has a left child, then the **left subtree** of *v* is the binary tree whose root is the left child of *v*, whose vertices consist of the left child of *v* and all its descendants, and whose edges consist of all those edges of *T* that connect the vertices of the left subtree.

The **right subtree** of *v* is defined analogously.

# Binary Trees

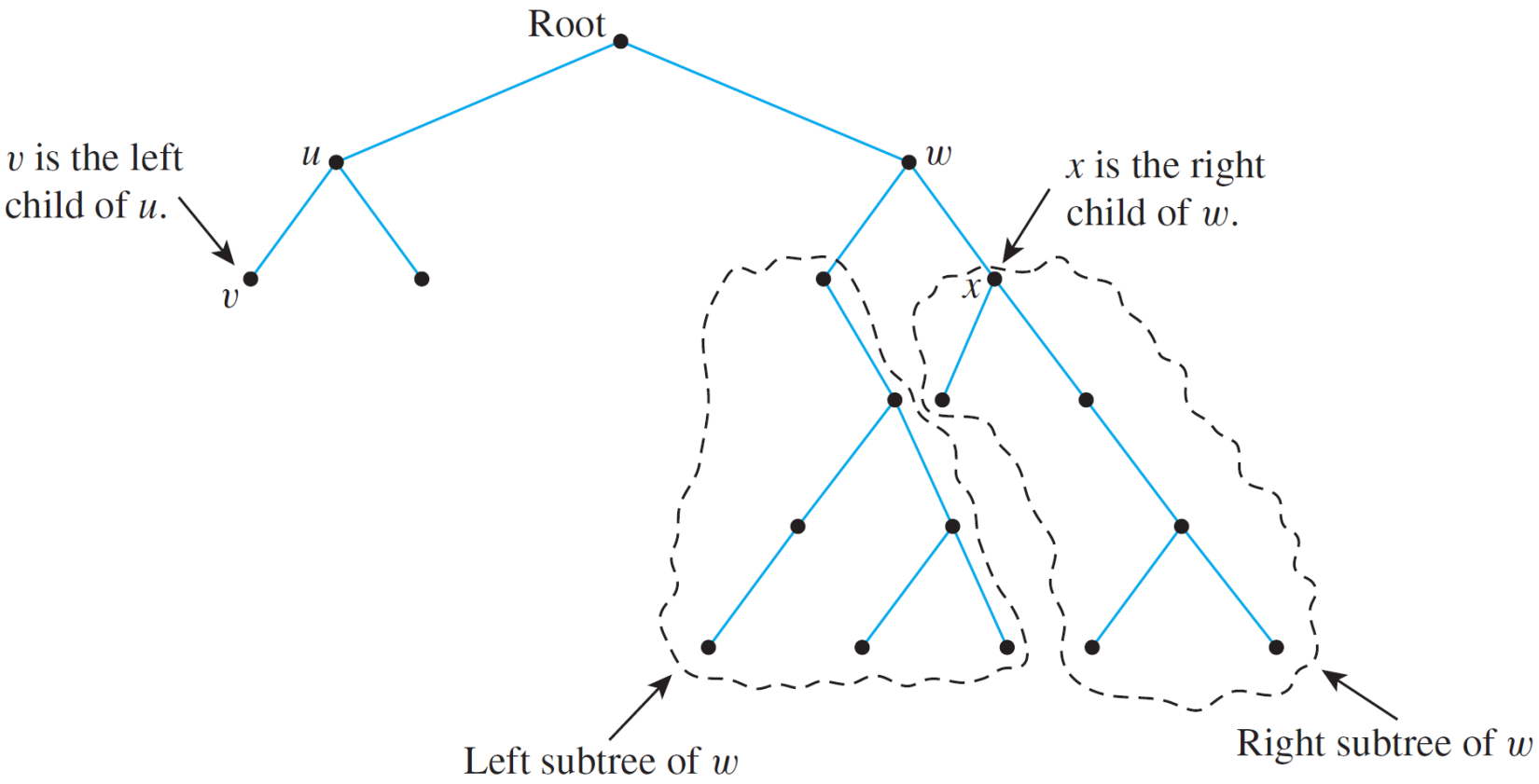
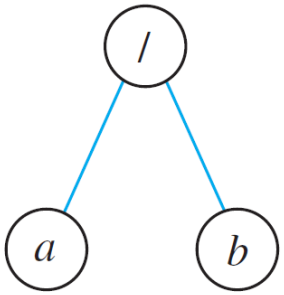



**Figure 10.6.2** A Binary Tree

# Example – Representation of Algebraic Expressions

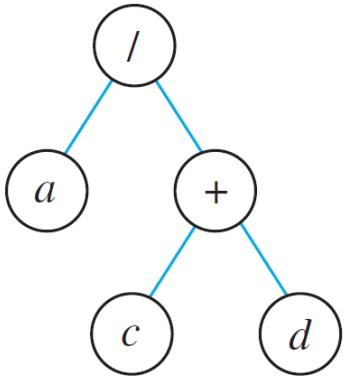
Binary trees are used in many ways in computer science. One use is to represent algebraic expressions with arbitrary nesting of balanced parentheses.

For instance, the following (labeled) binary tree represents the expression $a/b$: The operator is at the root and acts on the left and right children of the root in **left-right order**.



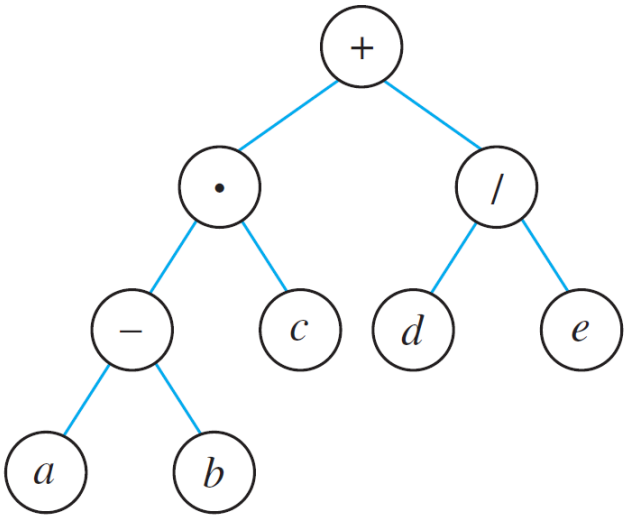
## Example – Representation of Algebraic Expressions

More generally, the binary tree shown below represents the expression $a/(c + d)$. In such a representation, the internal vertices are arithmetic operators, the terminal vertices are variables, and the operator at each vertex acts on its left and right subtrees in left-right order.



## Example – Representation of Algebraic Expressions

Draw a binary tree to represent the expression $((a - b) \cdot c) + (d/e)$.



# Binary Tree Traversal

Tree traversal (also known as tree search) is the process of visiting each node in a tree data structure exactly once in a systematic manner.

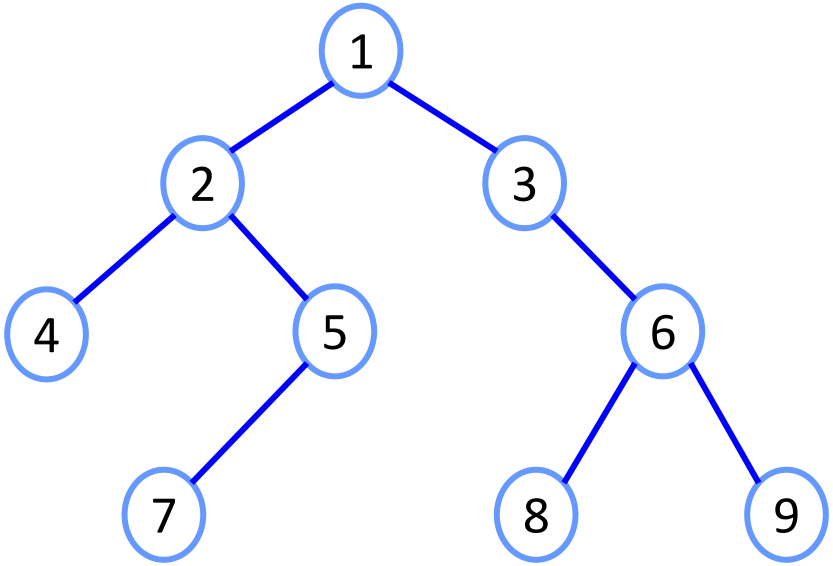
There are two types of traversal: breadth-first search (BFS) or depth-first search (DFS).

The following sections describe BFS and DFS on binary trees, but in general they can be applied on any type of trees, or even graphs.

# Breadth-First Search

In breadth-first search (by E.F. Moore), it starts at the root and visits its adjacent vertices, and then moves to the next level.

The figure shows the order of the vertices visited.

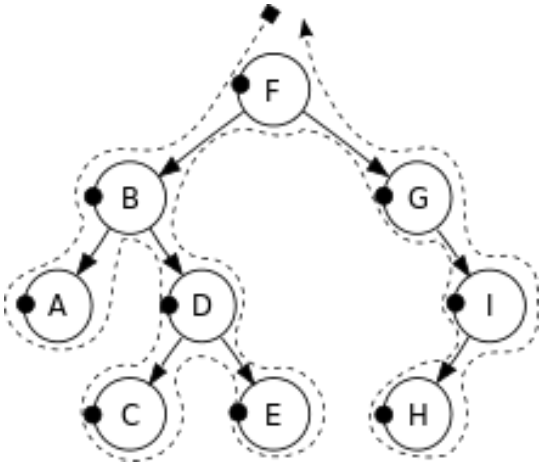


# Depth-First Search

There are three types of depth-first traversal:

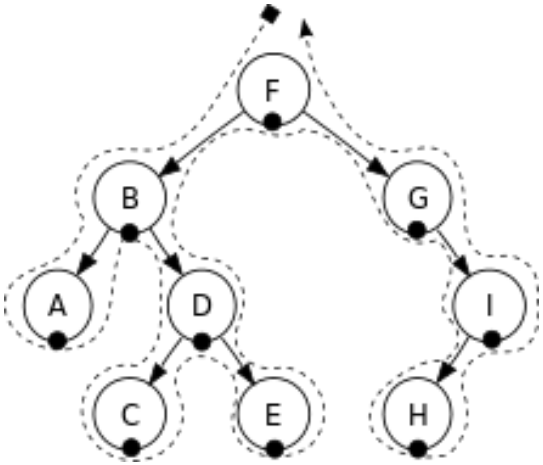
- Pre-order
  - Print the data of the root (or current vertex)
  - Traverse the left subtree by recursively calling the pre-order function
  - Traverse the right subtree by recursively calling the pre-order function
- In-order
  - Traverse the left subtree by recursively calling the in-order function
  - Print the data of the root (or current vertex)
  - Traverse the right subtree by recursively calling the in-order function
- Post-order
  - Traverse the left subtree by recursively calling the post-order function
  - Traverse the right subtree by recursively calling the post-order function
  - Print the data of the root (or current vertex)

# Depth-First Search



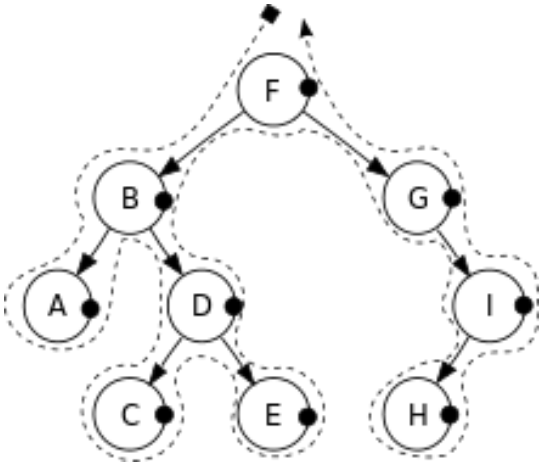

Pre-order:

F, B, A, D, C, E, G, I, H




In-order:

A, B, C, D, E, F, G, H, I




Post-order:

A, C, E, D, B, H, I, G, F

Acknowledgement: Wikipedia https://en.wikipedia.org/wiki/Tree_traversal

# 10.7 Spanning Trees and Shortest Paths

An East Coast airline company wants to expand service to the Midwest and has received permission from the Federal Aviation Authority to fly any of the routes shown in Figure 10.7.1.

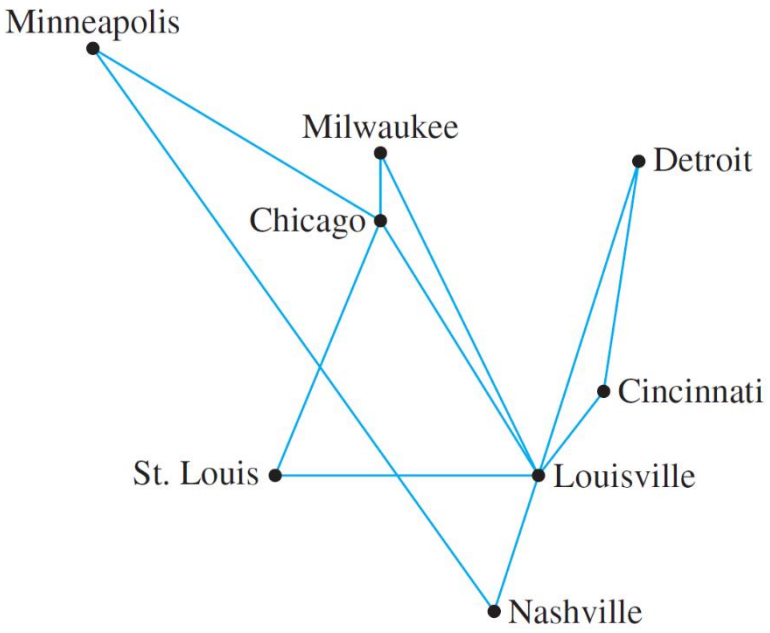



Figure 10.7.1

The company wishes to legitimately advertise service to all the cities shown but, for reasons of economy, wants to use the least possible number of individual routes to connect them. One possible route system is given in Figure 10.7.2, where the chosen routes are in red.

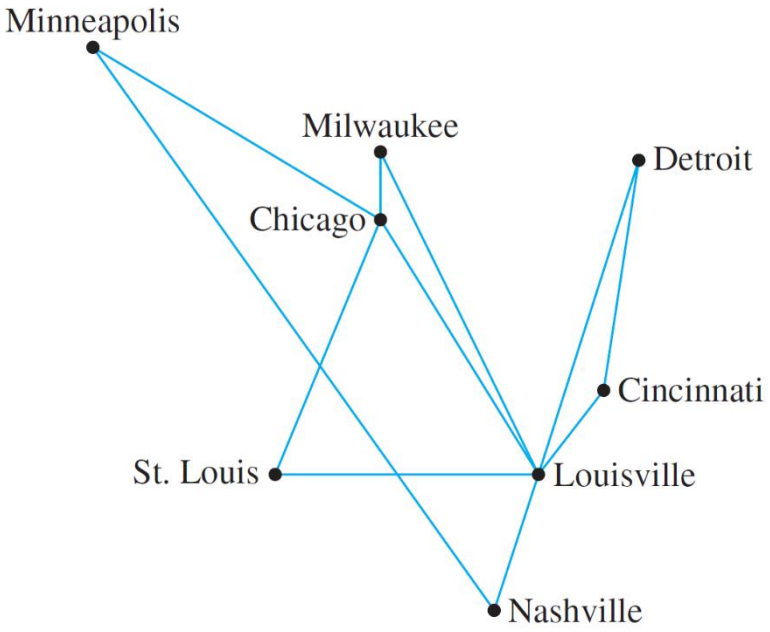



Figure 10.7.1

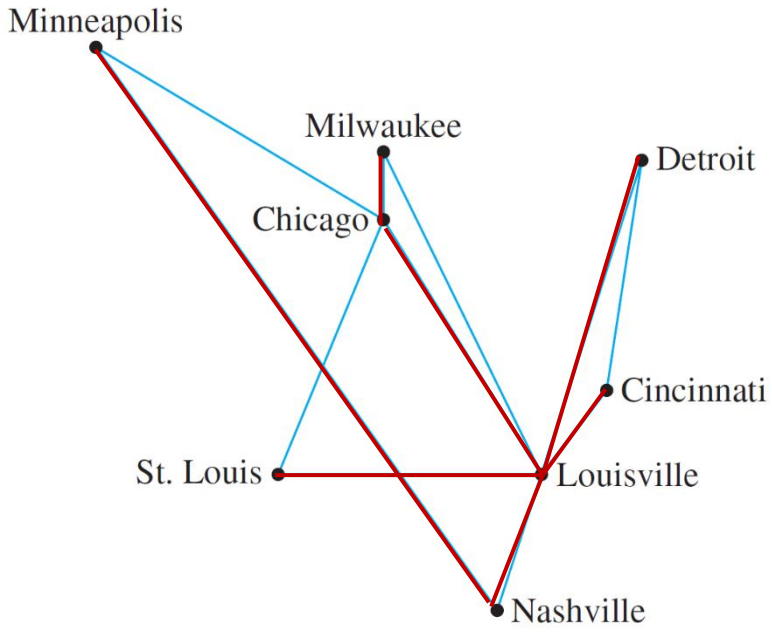



Figure 10.7.2

What you have seen is a spanning tree.

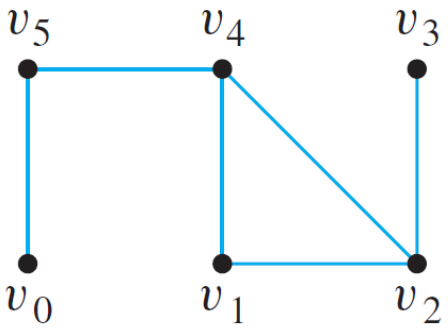
## Definition: Spanning Tree

A **spanning tree** for a graph $G$ is a subgraph of $G$ that contains every vertex of $G$ and is a tree.

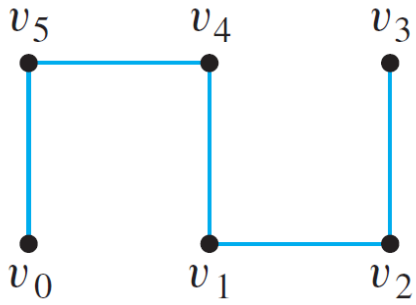
## Proposition 10.7.1

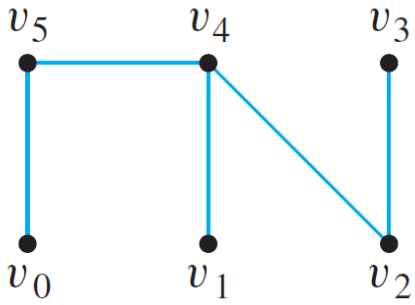
1. Every connected graph has a spanning tree.
2. Any two spanning trees for a graph have the same number of edges.

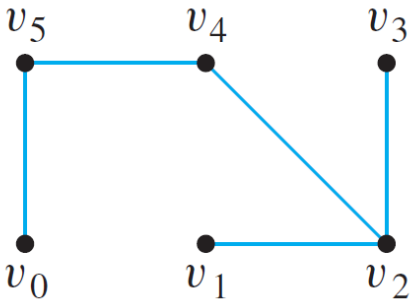
# Example: Find all spanning trees for the graph *G* below.




The graph *G* has one circuit $v_2v_1v_4v_2$ and removal of any edge of the circuit gives a tree. Hence there are three spanning trees for *G*.



# Minimum Spanning Trees

The graph of the routes allowed by the Federal Aviation Authority shown in Figure 10.7.1 can be annotated by adding the distances (in miles) between each pair of cities.

Now suppose the airline company wants to serve all the cities shown, but with a route system that minimizes the total mileage.

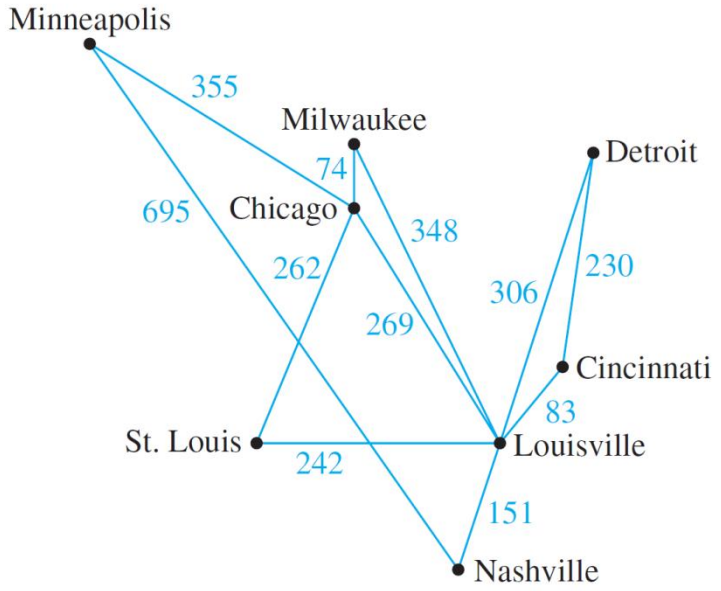



Figure 10.7.3

# Minimum Spanning Trees

## Definitions: Weighted Graph, Minimum Spanning Tree

A **weighted graph** is a graph for which each edge has an associated positive real number **weight** . The sum of the weights of all the edges is the **total weight** of the graph.

A **minimum spanning tree** for a connected weighted graph is a spanning tree that has the least possible total weight compared to all other spanning trees for the graph.

If *G* is a weighted graph and *e* is an edge of *G*, then ***w(e)*** denotes the weight of *e* and ***w(G)*** denotes the total weight of *G*.

# Kruskal's Algorithm (Joseph B. Kruskal, 1956)

In Kruskal's algorithm, the edges of a connected weighted graph are examined one by one in order of increasing weight.

At each stage the edge being examined is added to what will become the minimum spanning tree, provided that this addition does not create a circuit.

After $n - 1$ edges have been added (where $n$ is the number of vertices of the graph), these edges, together with the vertices of the graph, form a minimum spanning tree for the graph.

## Algorithm 10.7.1 Kruskal

Input: $G$ [a connected weighted graph with $n$ vertices]

Algorithm:

1. Initialize $T$ to have all the vertices of $G$ and no edges.
2. Let $E$ be the set of all edges of $G$, and let $m = 0$.
3. While ($m < n - 1$)
   - 3a. Find an edge $e$ in $E$ of least weight.
   - 3b. Delete $e$ from $E$.
   - 3c. If addition of $e$ to the edge set of $T$ does not produce a circuit, then add $e$ to the edge set of $T$ and set $m = m + 1$

   End while

Output: $T$ [$T$ is a minimum spanning tree for $G$]

Example: Describe the action of Kruskal's algorithm on the graph shown in Figure 10.7.4, where $n = 8$.

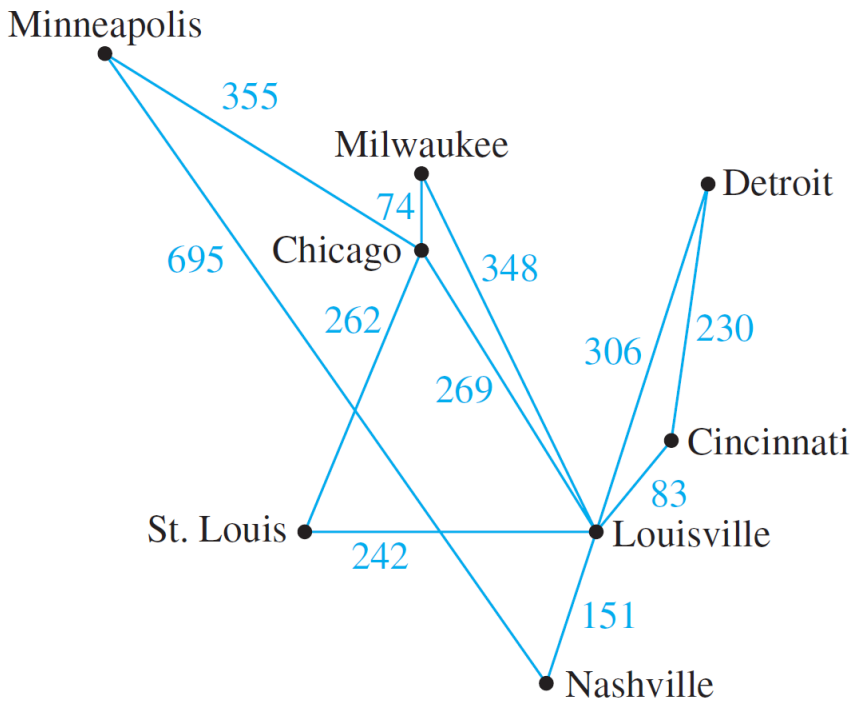



Figure 10.7.4

Using Kruskal's algorithm we can formulate the following table.

| | Edge considered | Wt | Action taken |
|---|---|---|---|
| 1 | → Chi – Mil | 74 | added |
| 2 | → Lou – Cin | 83 | added |
| 3 | → Lou – Nas | 151 | added |
| 4 | → Cin – Det | 230 | added |
| 5 | → StL – Lou | 242 | added |
| 6 | → StL – Chi | 262 | added |
| 7 | → Chi – Lou | 269 | not added |
| 8 | → Lou – Det | 306 | not added |
| 9 | → Lou – Mil | 348 | not added |
| 10 | → Min – Chi | 355 | added |

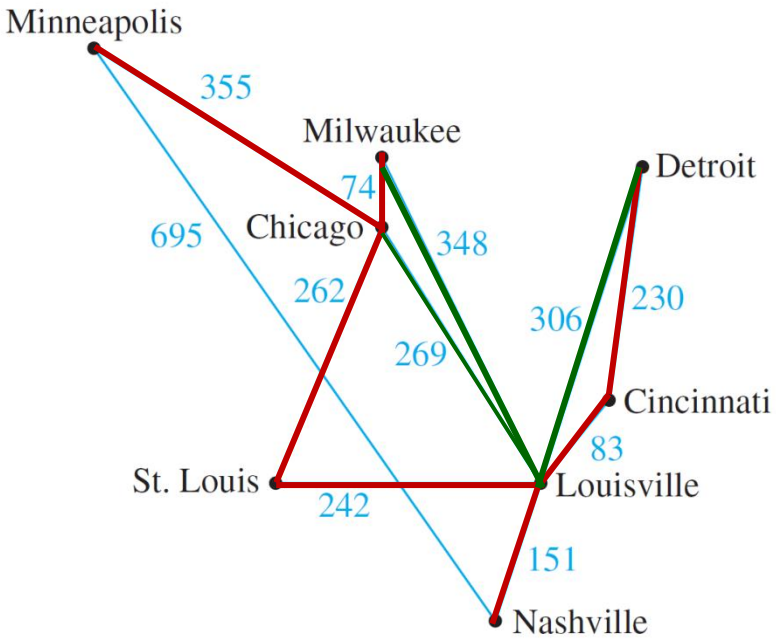
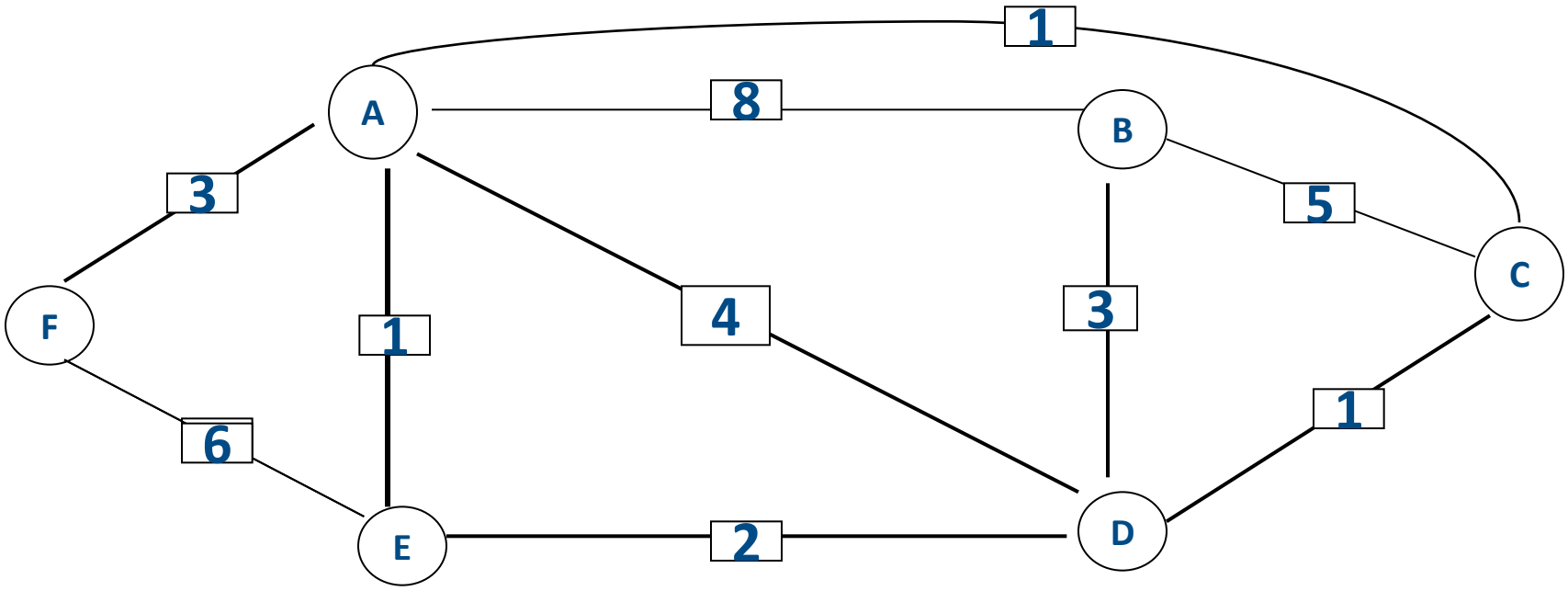


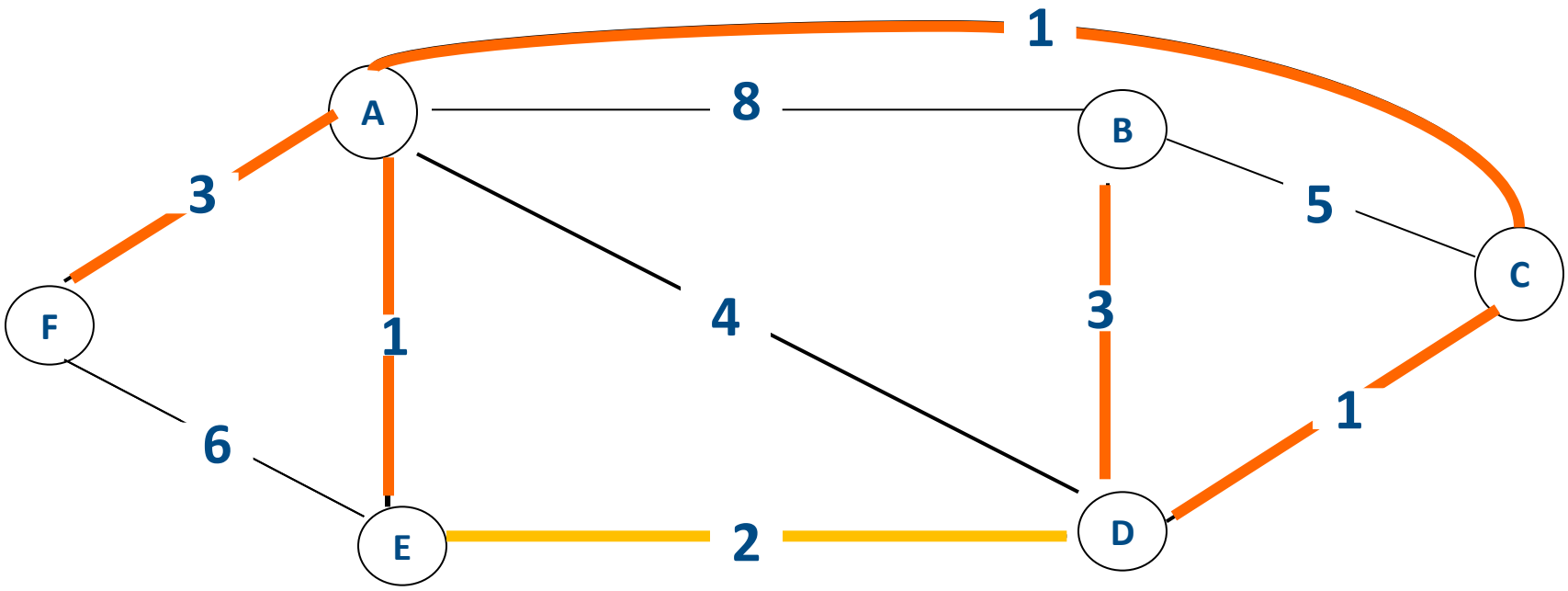

Figure 10.7.4

When Kruskal's algorithm is used on a graph in which some edges have the same weight as others, more than one minimum spanning tree can occur as output.

To make the output unique, the edges of the graph can be placed in an array and edges having the same weight can be added in the order they appear in the array.

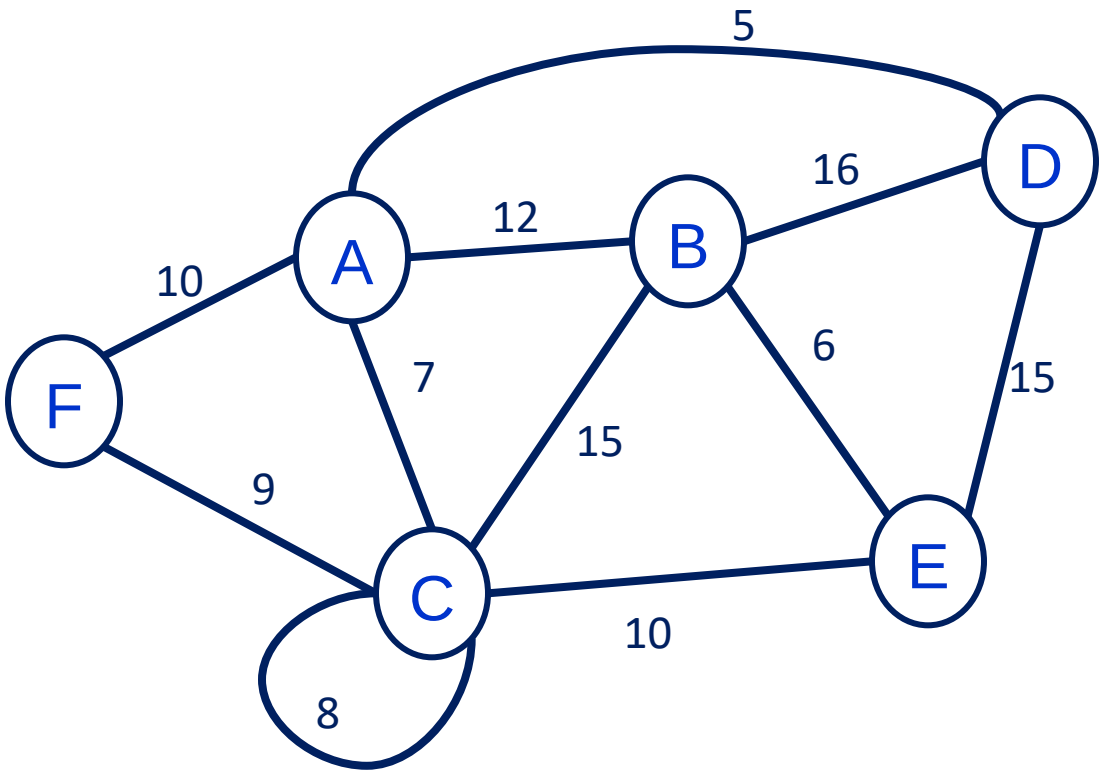

1
A
8
B
3
5
C
F
1
4
3
1
6
E
2
D

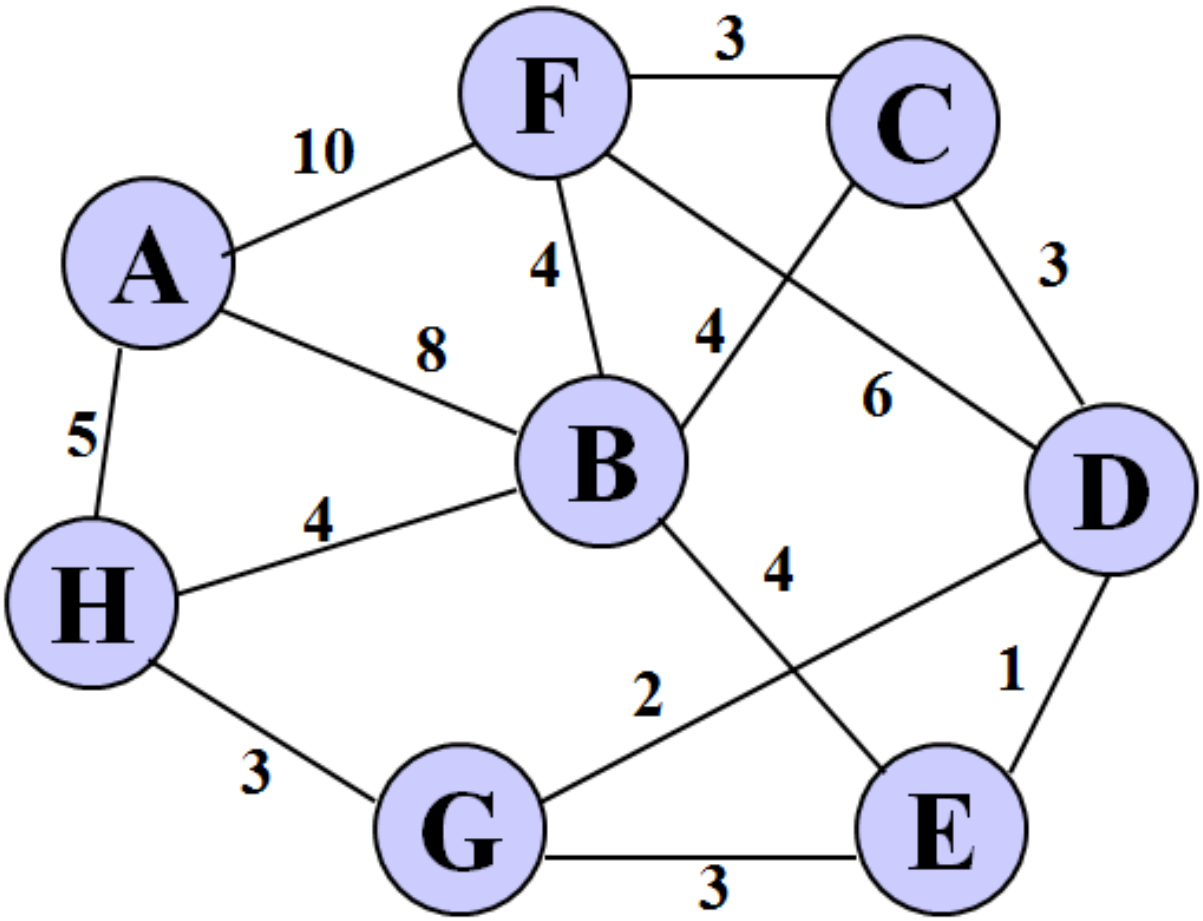
## Example




| AC | AE | CD | DE | AF | BD | AD | BC | EF | AB |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 6 | 8 |

T = { AC, AE, CD, AF, BD } is a minimum spanning tree


5
D
16
12
B
A
10
F
7
6
15
15
9
C
E
10
8


F
C
A
B
D
H
G
E
3
10
4
3
8
4
6
5
4
4
2
1
3
3

# Prim's Algorithm (Robert C. Prim, 1957)

Prim's algorithm works differently from Kruskal's. It builds a minimum spanning tree *T* by expanding outward in connected links from some vertex.

One edge and one vertex are added at each stage. The edge added is the one of least weight that connects the vertices already in *T* with those not in *T*, and the vertex is the endpoint of this edge that is not already in *T*.

## Algorithm 10.7.2 Prim

Input: $G$ [a connected weighted graph with $n$ vertices]

Algorithm:

1. Pick a vertex $v$ of $G$ and let $T$ be the graph with this vertex only.
2. Let $V$ be the set of all vertices of $G$ except $v$.
3. For $i = 1$ to $n - 1$
   - 3a. Find an edge $e$ of $G$ such that (1) $e$ connects $T$ to one of the vertices in $V$, and (2) $e$ has the least weight of all edges connecting $T$ to a vertex in $V$. Let $w$ be the endpoint of $e$ that is in $V$.
   - 3b. Add $e$ and $w$ to the edge and vertex sets of $T$, and delete $w$ from $V$.

Output: $T$ [$T$ is a minimum spanning tree for $G$]

Example: Describe the action of Prim's algorithm on the graph shown in Figure 10.7.6, using the Minneapolis vertex as a starting point.

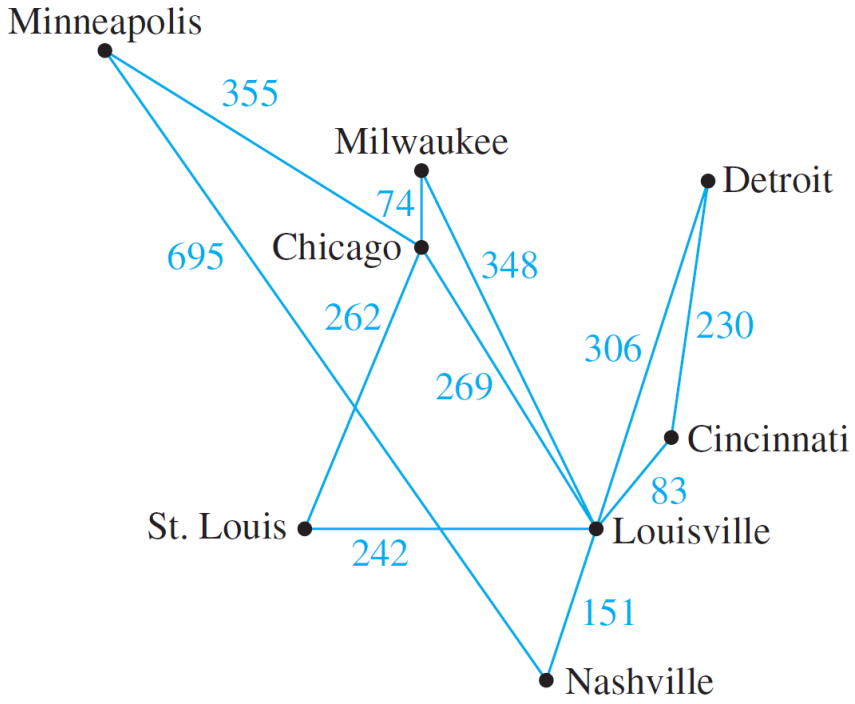



Figure 10.7.6

Using Prim's algorithm we can formulate the following table.

| | Vertex added | Edge added | Weight |
|---|---|---|---|
| 0 | Minneapolis | | |
| 1 | Chicago | Min – Chi | 355 |
| 2 | Milwaukee | Chi – Mil | 74 |
| 3 | St. Louis | Chi – StL | 262 |
| 4 | Louisville | StL – Lou | 242 |
| 5 | Cincinnati | Lou – Cin | 83 |
| 6 | Nashville | Lou – Nas | 151 |
| 7 | Detroit | Cin – Det | 230 |

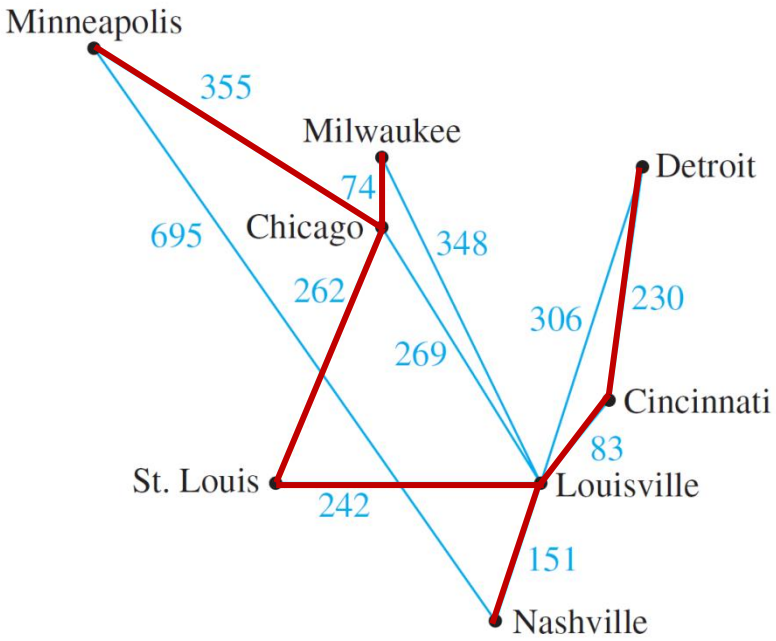



Figure 10.7.6