

$g_n = nC$. Then we have $nC - 6(n-1)C + 5(n-2)C = 3$, which simplifies to $6C - 10C = 3$ so that $C = -3/4$. Thus, $g_n = -3n/4$. This won't be enough if 1 is a root of multiplicity 2, that is, if $r - 1$ is a factor of the characteristic polynomial. Then there is a particular solution of the form $g_n = Cn^2$. For second-order equations, you never have to go past this. If the right-hand side is a polynomial of degree greater than 0, then the process works just the same, except that you start with a polynomial of the same degree, increase the degree by 1, if necessary, and then once more, if need be. For example, if the right-hand side were $f_n = 2n - 1$, we would start by guessing a particular solution $g_n = C_1n + C_2$. If it turned out that 1 was a characteristic root, we would amend our guess to $g_n = C_1n^2 + C_2n + C_3$. If 1 is a double root, this will fail also, but $g_n = C_1n^3 + C_2n^2 + C_3n + C_4$ will work in this case.

Another case where there is a simple way of guessing a particular solution is when the right-hand side is an exponential, say $f_n = Cn$. In that case, we guess that a particular solution is just a constant multiple of f , say $(g)_n = kCn$. Again, we gave trouble when 1 is a characteristic root. We then guess that $g_n = kn^2Cn$, which will fail only if 1 is a double root. In that case we must use $g_n = kn^3Cn$, which is as far as we ever have to go in the second-order case. These same ideas extend to higher-order recurrence relations, but we usually solve them numerically, rather than exactly. A third-order linear difference equation with constant coefficients leads to a cubic characteristic polynomial. There is a formula for the roots of a cubic, but it's very complicated.

For fourth-degree polynomials, there's also a formula, but it's even worse. For fifth and higher degrees, no such formula exists. Even for the third-order case, the exact solution of a simple-looking inhomogeneous linear recurrence relation with constant coefficients can take pages to write down. The coefficients will be complicated expressions involving square roots and cube roots. For most, if not all, purposes, a simpler answer with numerical coefficients is better, even though they must in the nature of things, be approximate.

The procedure I've suggested may strike you as silly. After all, we've already solved the characteristic equation, so we know whether 1 is a characteristic root, and what its multiplicity is. Why not start with a polynomial of the correct degree? This is all well and good, while you're taking the course, and remember the procedure in detail. However, if you have to use this procedure some years from now, you probably won't remember all the details. Then the method I've suggested will be valuable. Alternatively, you can start with a general polynomial of the maximum possible degree. This leads to a lot of extra work if you're solving by hand, but it's the approach I prefer for computer solution.

UNIT V

Graph Theory

Representation of Graphs:

There are two different sequential representations of a graph. They are

- Adjacency Matrix representation
- Path Matrix representation

Adjacency Matrix Representation

Suppose G is a simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1, v_2, \dots, v_m$. Then the adjacency matrix $A = (a_{ij})$ of the graph G is the $m \times m$ matrix defined as follows:

$a_{ij} = 1$ if v_i is adjacent to v_j , that is, if there is an edge (v_i, v_j)
 $a_{ij} = 0$ otherwise

Suppose G is an undirected graph. Then the adjacency matrix A of G will be a symmetric matrix, i.e., one in which $a_{ij} = a_{ji}$; for every i and j .

Drawbacks

12. It may be difficult to insert and delete nodes in G .
13. If the number of edges is $O(m)$ or $O(m \log^2 m)$, then the matrix A will be sparse, hence a great deal of space will be wasted.

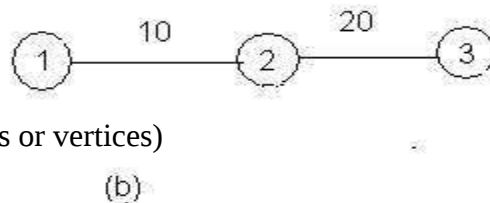
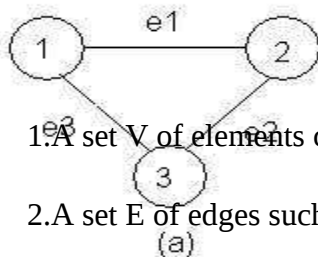
Path Matrix Representation

Let G be a simple directed graph with m nodes, $v_1, v_2, \dots, v_m$. The path matrix of G is the m -square matrix $P = (p_{ij})$ defined as follows:

$p_{ij} = 1$ if there is a path from v_i to v_j
 $p_{ij} = 0$ otherwise

Graphs and Multigraphs

A graph G consists of two things:



1. A set V of elements called nodes (or points or vertices)

2. A set E of edges such that each edge e in E is identified with a unique

(unordered) pair $[u, v]$ of nodes in V , denoted by $e = [u, v]$

Sometimes we indicate the parts of a graph by writing $G = (V, E)$.

Suppose $e = [u, v]$. Then the nodes u and v are called the endpoints of e , and u and v are said to be adjacent nodes or neighbors. The degree of a node u , written $\deg(u)$, is the number of edges containing u . If $\deg(u) = 0$ — that is, if u does not belong to any edge — then u is called an isolated node.

Path and Cycle

A path P of length n from a node u to a node v is defined as a sequence of $n + 1$ nodes. $P = (v_0, v_1, v_2, \dots, v_n)$ such that $u = v_0$; v_{i-1} is adjacent to v_i for $i = 1, 2, \dots, n$ and $v_n = v$.

Types of Path

1. Simple Path
2. Cycle Path

(i) Simple Path

Simple path is a path in which first and last vertex are different ($V_0 \neq V_n$)

(ii) Cycle Path

Cycle path is a path in which first and last vertex are same ($V_0 = V_n$). It is also called as Closed path.

Connected Graph

A graph G is said to be connected if there is a path between any two of its nodes.

Complete Graph

A graph G is said to be complete if every node u in G is adjacent to every other node v in G .

Tree

A connected graph T without any cycles is called a tree graph or free tree or, simply, a tree.

Labeled or Weighted Graph

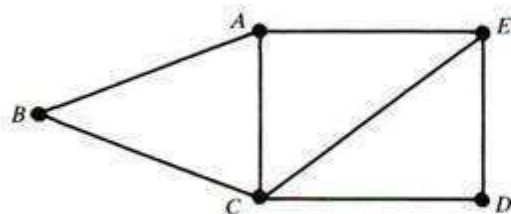
If the weight is assigned to each edge of the graph then it is called as Weighted or Labeled graph.

The definition of a graph may be generalized by permitting the following:

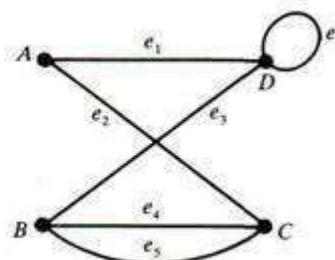
■ **Multiple edges:** Distinct edges e and e' are called multiple edges if they connect the same endpoints, that is, if $e = [u, v]$ and $e' = [u, v]$.

■ **Loops:** An edge e is called a loop if it has identical endpoints, that is, if $e = [u, u]$.

■ **Finite Graph:** A multigraph M is said to be finite if it has a finite number of nodes and a finite number of edges.



(a) Graph.



(b) Multigraph.

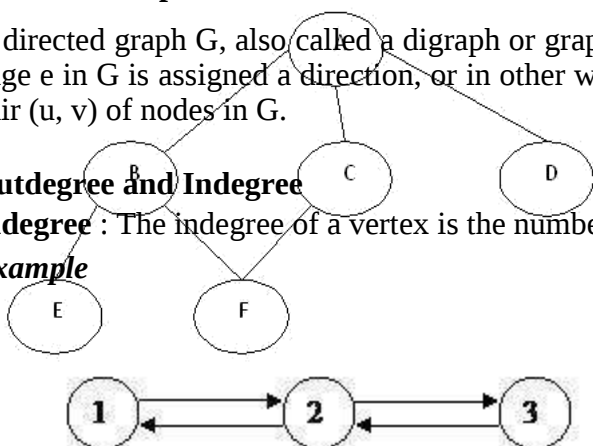
Directed Graphs

A directed graph G , also called a digraph or graph is the same as a multigraph except that each edge e in G is assigned a direction, or in other words, each edge e is identified with an ordered pair (u, v) of nodes in G .

Outdegree and Indegree

Indegree : The indegree of a vertex is the number of edges for which v is head

Example

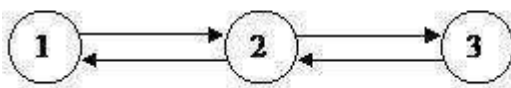


Indegree of 1 = 1

Indegree of 2 = 2

Outdegree : The outdegree of a node or vertex is the number of edges for which v is tail.

Example



Outdegree of 1 = 1

Outdegree of 2 = 2

Simple Directed Graph

A directed graph G is said to be simple if G has no parallel edges. A simple graph G may have loops, but it cannot have more than one loop at a given node.

Graph Traversal

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.

Depth First Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph:

www.JntukMaterials.com

As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be “A B E F C D”. DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

6. **Step 1:** Push the root node in the Stack.
7. **Step 2:** Loop until stack is empty.
8. **Step 3:** Peek the node of the stack.
9. **Step 4:** If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
10. **Step 5:** If the node does not have any unvisited child nodes, pop the node from the stack.

Based upon the above steps, the following Java code shows the implementation of the DFS algorithm:

```
public void dfs()
{
    //DFS uses Stack data structure

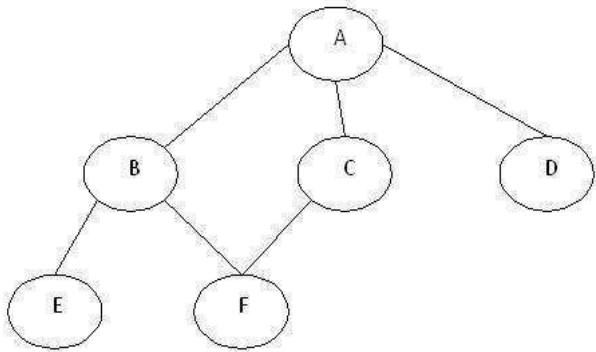
    Stack s=new Stack();
    s.push(this.rootNode);
    rootNode.visited=true;
    printNode(rootNode);
    while(!s.isEmpty())
    {
        Node n=(Node)s.peek();

        Node child=getUnvisitedChildNode(n);
        if(child!=null)
        {
            child.visited=true;
            printNode(child);
            s.push(child);
        }
        else
        {
            s.pop();
        }
    }

    //Clear visited property
    of nodes clearNodes();
}
```

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "A B C D E F". The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

1. **Step 1:** Push the root node in the Queue.
2. **Step 2:** Loop until the queue is empty.
3. **Step 3:** Remove the node from the Queue.
4. **Step 4:** If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Based upon the above steps, the following Java code shows the implementation of the BFS algorithm:

```
public void bfs()
{
    //BFS uses Queue data structure

    Queue q=new LinkedList();
    q.add(this.rootNode);
    printNode(this.rootNode);
    rootNode.visited=true;
    while(!q.isEmpty())
    {
        Node n=(Node)q.remove();
        Node child=null;
        while((child=getUnvisitedChildNode(n))!=null)
        {
            child.visited=true;
            printNode(child);
            q.add(child);
        }
    }
}
```

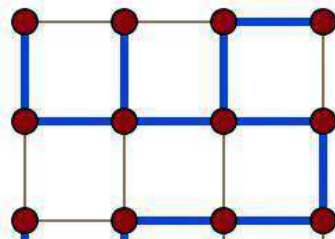
```
    }  
}  
  
//Clear visited property  
of nodes clearNodes();  
}
```

Spanning Trees:

In the mathematical field of graph theory, a **spanning tree** T of a connected, undirected graph G is a tree composed of all the vertices and some (or perhaps all) of the edges of G . Informally, a spanning tree of G is a selection of edges of G that form a tree *spanning* every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of G must belong to T .

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

Example:



A spanning tree (blue heavy edges) of a grid graph.

Spanning forests

A **spanning forest** is a type of subgraph that generalises the concept of a spanning tree. However, there are two definitions in common use. One is that a spanning forest is a subgraph that consists of a spanning tree in each connected component of a graph. (Equivalently, it is a maximal cycle-free subgraph.) This definition is common in computer science and optimisation. It is also the definition used when discussing minimum spanning forests, the generalization to disconnected graphs of minimum spanning trees. Another definition, common in graph theory, is that a spanning forest is any subgraph that is both a forest (contains no cycles) and spanning (includes every vertex).

Counting spanning trees

The number $t(G)$ of spanning trees of a connected graph is an important invariant. In some cases, it is easy to calculate $t(G)$ directly. It is also widely used in data structures in different computer languages. For example, if G is itself a tree, then $t(G)=1$, while if G is the cycle graph C_n with n vertices, then $t(G)=n$. For any graph G , the number $t(G)$ can be calculated using Kirchhoff's matrix-tree theorem (follow the link for an explicit example using the theorem).

Cayley's formula is a formula for the number of spanning trees in the complete graph K_n with n vertices. The formula states that $t(K_n) = n^{n-2}$. Another way of stating Cayley's formula is that there are exactly n^{n-2} labelled trees with n vertices. Cayley's formula can be proved using Kirchhoff's matrix-tree theorem or via the Prüfer code.

If G is the complete bipartite graph $K_{p,q}$, then $t(G) = p^{q-1} q^{p-1}$, while if G is the n -dimensional hypercube graph Q_n , then $t(G) = 2^{2^n - n - 1} \prod_{k=2}^n k^{\binom{n}{k}}$. These formulae are also consequences of the matrix-tree theorem.

If G is a multigraph and e is an edge of G , then the number $t(G)$ of spanning trees of G satisfies the *deletion-contraction recurrence* $t(G) = t(G-e) + t(G/e)$, where $G-e$ is the multigraph obtained by deleting e and G/e is the contraction of G by e , where multiple edges arising from

this contraction are not deleted.

Uniform spanning trees

A spanning tree chosen randomly from among all the spanning trees with equal probability is called a uniform spanning tree (UST). This model has been extensively researched in probability and mathematical physics.

Algorithms

The classic spanning tree algorithm, depth-first search (DFS), is due to Robert Tarjan. Another important algorithm is based on breadth-first search (BFS).

Planar Graphs:

In graph theory, a **planar graph** is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.

A planar graph already drawn in the plane without edge intersections is called a **plane graph** or **planar embedding of the graph**. A plane graph can be defined as a planar graph with a mapping from every node to a point in 2D space, and from every edge to a plane curve, such that the extreme points of each curve are the points mapped from its end nodes, and all curves are disjoint except on their extreme points. Plane graphs can be encoded by combinatorial maps.

It is easily seen that a graph that can be drawn on the plane can be drawn on the sphere as well, and vice versa.

The equivalence class of topologically equivalent drawings on the sphere is called a **planar map**. Although a plane graph has an **external** or **unbounded** face, none of the faces of a planar map have a particular status.

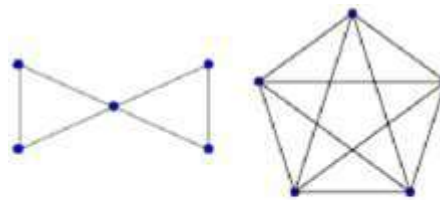
Applications

- Telecommunications – e.g. spanning trees
- Vehicle routing – e.g. planning routes on roads without underpasses
- VLSI – e.g. laying out circuits on computer chip.
- The puzzle game Planarity requires the player to "untangle" a planar graph so that none of its edges intersect.

Example graphs

planar

non planar



Butterfly graph

K5

Graph Theory and Applications:

Graphs are among the most ubiquitous models of both natural and human-made structures. They can be used to model many types of relations and process dynamics in physical, biological and social systems. Many problems of practical interest can be represented by graphs.

In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. One practical example: The link structure of a website could be represented by a directed graph. The vertices are the web pages available at the website and a directed edge from page *A* to page *B* exists if and only if *A* contains a link to *B*. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. There, the transformation of graphs is often formalized and represented by graph rewrite systems. They are either directly used or properties of the rewrite systems (e.g. confluence) are studied. Complementary to graph transformation systems focussing on rule-based in-memory manipulation of graphs are graph databases geared towards transaction-safe, persistent storing and querying of graph-structured data.

Graph-theoretic methods, in various forms, have proven particularly useful in linguistics, since natural language often lends itself well to discrete structure. Traditionally, syntax and compositional semantics follow tree-based structures, whose expressive power lies in the Principle of Compositionality, modeled in a hierarchical graph. Within lexical semantics, especially as applied to computers, modeling word meaning is easier when a given word is understood in terms of related words; semantic networks are therefore important in computational linguistics. Still other methods in phonology (e.g. Optimality Theory, which uses lattice graphs) and morphology (e.g. finite-state morphology, using finite-state transducers) are common in the analysis of language as a graph. Indeed, the usefulness of this area of mathematics to linguistics has borne organizations such as TextGraphs, as well as various 'Net' projects, such as WordNet, VerbNet, and others.

Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms. For example, Franzblau's shortest-path (SP) rings. In chemistry a graph makes a natural model for a molecule, where vertices represent atoms and edges bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching. In statistical physics, graphs can represent local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems.

Graph theory is also widely used in sociology as a way, for example, to measure actors' prestige or to explore diffusion mechanisms, notably through the use of social network analysis software. Likewise, graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist (or habitats) and the edges represent migration paths, or movement between the regions. This information is important when looking at breeding patterns or tracking the spread of disease, parasites or how changes to the movement can affect other species.

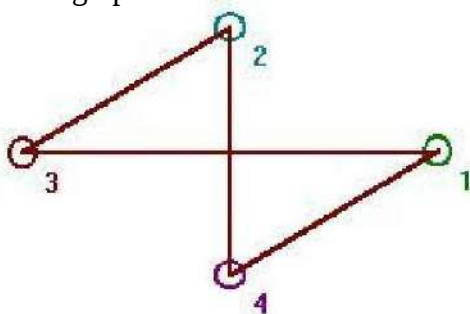
In mathematics, graphs are useful in geometry and certain parts of topology, e.g. Knot Theory. Algebraic graph theory has close links with group theory.

A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pairwise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road.

Basic Concepts Isomorphism:

Let G_1 and G_2 be two graphs and let f be a function from the vertex set of G_1 to the vertex set of G_2 . Suppose that f is one-to-one and onto & $f(v)$ is adjacent to $f(w)$ in G_2 if and only if v is adjacent to w in G_1 .

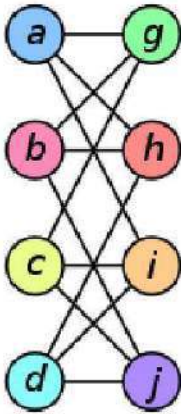
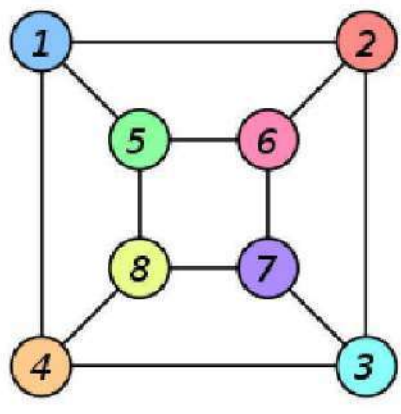
Then we say that the function f is an isomorphism and that the two graphs G_1 and G_2 are isomorphic. So two graphs G_1 and G_2 are isomorphic if there is a one-to-one correspondence between vertices of G_1 and those of G_2 with the property that if two vertices of G_1 are adjacent then so are their images in G_2 . If two graphs are isomorphic then as far as we are concerned they are the same graph though the location of the vertices may be different. To show you how the program can be used to explore isomorphism draw the graph in figure 4 with the program (first get the null graph on four vertices and then use the right mouse to add edges).



Save this graph as Graph 1 (you need to click Graph then Save). Now get the circuit graph with 4 vertices. It looks like figure 5, and we shall call it C(4).

Example:

The two graphs shown below are isomorphic, despite their different looking drawings.

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

Subgraphs:

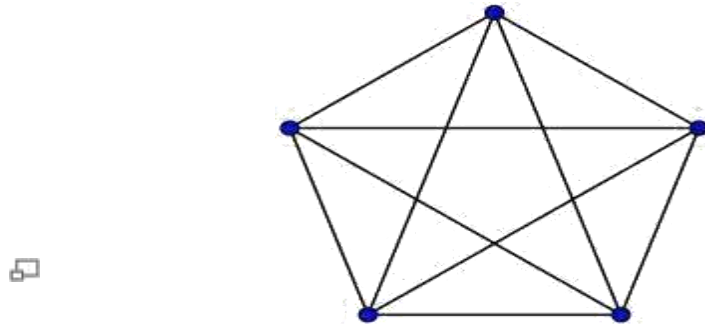
A **subgraph** of a graph G is a graph whose vertex set is a subset of that of G , and whose adjacency relation is a subset of that of G restricted to this subset. In the other direction, a **supergraph** of a graph G is a graph of which G is a subgraph. We say a graph G **contains** another graph H if some subgraph of G is H or is isomorphic to H .

A subgraph H is a **spanning subgraph**, or **factor**, of a graph G if it has the same vertex set as G . We say H spans G .

A subgraph H of a graph G is said to be **induced** if, for any pair of vertices x and y of H , xy is an edge of H if and only if xy is an edge of G . In other words, H is an induced subgraph of G if it has all the edges that appear in G over the same vertex set. If the vertex set of H is the subset S of $V(G)$, then H can be written as $G[S]$ and is said to be **induced by S** .

A graph that does *not* contain H as an induced subgraph is said to be **H -free**.

A **universal graph** in a class K of graphs is a simple graph in which every element in K can be embedded as a subgraph.



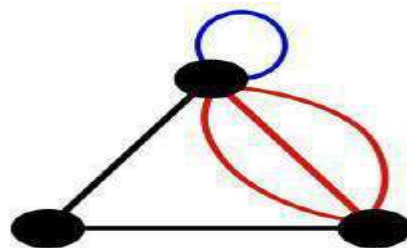
K_5 , a complete graph. If a subgraph looks like this, the vertices in that subgraph form a clique of size 5.

Multi graphs:

In mathematics, a **multigraph** or **pseudograph** is a graph which is permitted to have multiple edges, (also called "parallel edges"), that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge. Formally, a multigraph G is an ordered pair $G := (V, E)$ with

- V a set of *vertices* or *nodes*,
- E a multiset of unordered pairs of vertices, called *edges* or *lines*.

Multigraphs might be used to model the possible flight connections offered by an airline. In this case the multigraph would be a directed graph with pairs of directed parallel edges connecting cities to show that it is possible to fly both *to* and *from* these locations.



A multigraph with multiple edges (red) and a loop (blue). Not all authors allow multigraphs to have loops.

Euler circuits:

In graph theory, an **Eulerian trail** is a trail in a graph which visits every edge exactly once. Similarly, an **Eulerian circuit** is an Eulerian trail which starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. Mathematically the problem can be stated like this:

Given the graph on the right, is it possible to construct a path (or a cycle, i.e. a path starting and ending on the same vertex) which visits each edge exactly once?

Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published in 1873 by Carl Hierholzer.

The term **Eulerian graph** has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.

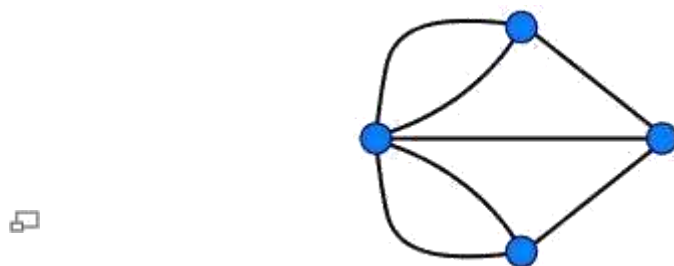
For the existence of Eulerian trails it is necessary that no more than two vertices have an odd degree; this means the Königsberg graph is *not* Eulerian. If there are no vertices of odd degree, all Eulerian trails are circuits. If there are exactly two vertices of odd degree, all Eulerian trails start at one of them and end at the other. Sometimes a graph that has an Eulerian trail but not an Eulerian circuit is called **semi-Eulerian**.

An **Eulerian trail**, **Eulerian trail** or **Euler walk** in an undirected graph is a path that uses each edge exactly once. If such a path exists, the graph is called **traversable** or **semi-eulerian**.

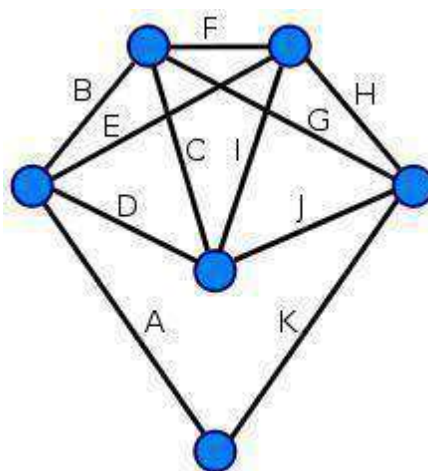
An **Eulerian cycle**, **Eulerian circuit** or **Euler tour** in an undirected graph is a cycle that uses each edge exactly once. If such a cycle exists, the graph is called **unicursal**. While such graphs are Eulerian graphs, not every Eulerian graph possesses an Eulerian cycle.

For directed graphs path has to be replaced with directed path and cycle with directed cycle.

The definition and properties of Eulerian trails, cycles and graphs are valid for multigraphs as well.



This graph is not Eulerian, therefore, a solution does not exist.



Every vertex of this graph has an even degree, therefore this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

Hamiltonian graphs:

In the mathematical field of graph theory, a **Hamiltonian path** (or **traceable path**) is a path in an undirected graph which visits each vertex exactly once. A **Hamiltonian cycle** (or **Hamiltonian circuit**) is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem which is NP-complete.

Hamiltonian paths and cycles are named after William Rowan Hamilton who invented the Icosian game, now also known as *Hamilton's puzzle*, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron. Hamilton solved this problem using the Icosian Calculus, an algebraic structure based on roots of unity with many similarities to the quaternions (also invented by Hamilton). This solution does not generalize to arbitrary graphs.

A *Hamiltonian path* or *traceable path* is a path that visits each vertex exactly once. A graph that contains a Hamiltonian path is called a **traceable graph**. A graph is **Hamilton-connected** if for every pair of vertices there is a Hamiltonian path between the two vertices.

A *Hamiltonian cycle*, *Hamiltonian circuit*, *vertex tour* or *graph cycle* is a cycle that visits each vertex exactly once (except the vertex which is both the start and end, and so is visited twice). A graph that contains a Hamiltonian cycle is called a **Hamiltonian graph**.

Similar notions may be defined for *directed graphs*, where each edge (arc) of a path or cycle can only be traced in a single direction (i.e., the vertices are connected with arrows and the edges traced "tail-to-head").

A **Hamiltonian decomposition** is an edge decomposition of a graph into Hamiltonian circuits.

Examples

- a complete graph with more than two vertices is Hamiltonian
- every cycle graph is Hamiltonian
- every tournament has an odd number of Hamiltonian paths
- every platonic solid, considered as a graph, is Hamiltonian

Chromatic Numbers:

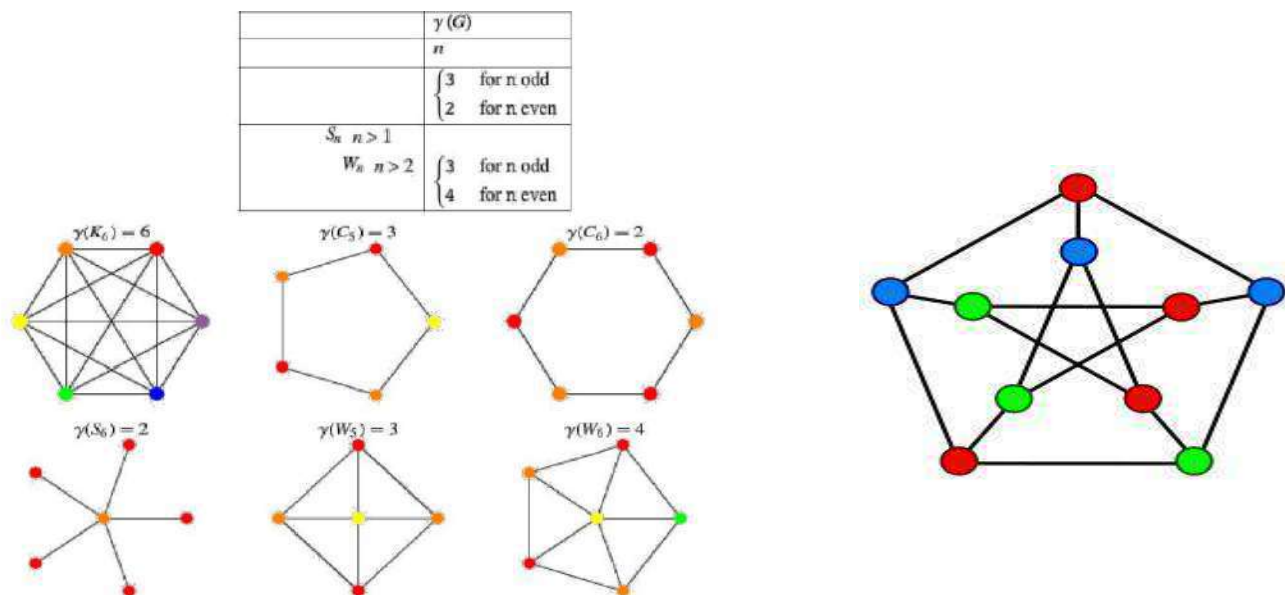
In graph theory, **graph coloring** is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a **vertex coloring**. Similarly, an **edge coloring** assigns a color to each edge so that no two adjacent edges share the same color, and a **face coloring** of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.

Vertex coloring is the starting point of the subject, and other coloring problems can be transformed into a vertex version. For example, an edge coloring of a graph is just a vertex coloring of its line graph, and a face coloring of a planar graph is just a vertex coloring of its planar dual. However, non-vertex coloring problems are often stated and studied *as is*. That is partly for perspective, and partly because some problems are best studied in non-vertex form, as for instance is edge coloring.

The convention of using colors originates from coloring the countries of a map, where each face is literally colored. This was generalized to coloring the faces of a graph embedded in the plane. By planar duality it became coloring the vertices, and in this form it generalizes to all graphs. In mathematical and computer representations it is typical to use the first few positive or nonnegative integers as the "colors". In general one can use any finite set as the "color set". The nature of the coloring problem depends on the number of colors but not on what they are.

Graph coloring enjoys many practical applications as well as theoretical challenges. Beside the classical types of problems, different limitations can also be set on the graph, or on the way a

color is assigned, or even on the color itself. It has even reached popularity with the general public in the form of the popular number puzzle Sudoku. Graph coloring is still a very active field of research.



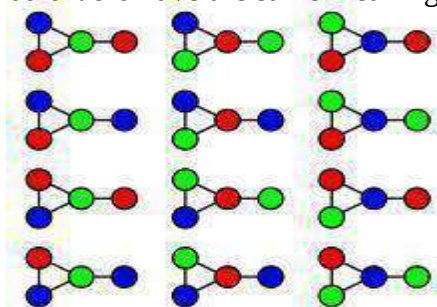
A proper vertex coloring of the Petersen graph with 3 colors, the minimum number possible.

Vertex coloring

When used without any qualification, a **coloring** of a graph is almost always a *proper vertex coloring*, namely a labelling of the graph's vertices with colors such that no two vertices sharing the same edge have the same color. Since a vertex with a loop could never be properly colored, it is understood that graphs in this context are loopless.

The terminology of using *colors* for vertex labels goes back to map coloring. Labels like *red* and *blue* are only used when the number of colors is small, and normally it is understood that the labels are drawn from the integers $\{1, 2, 3, \dots\}$.

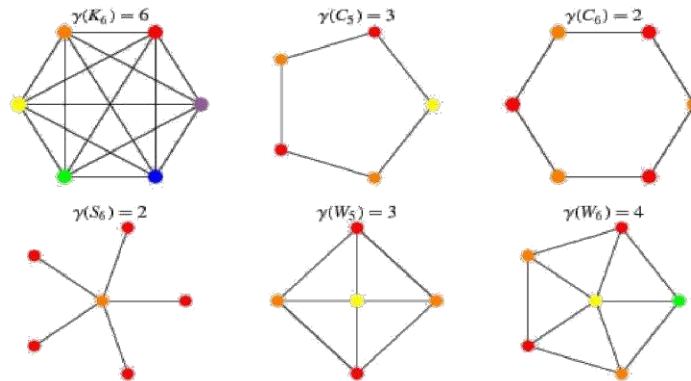
A coloring using at most k colors is called a (proper) **k -coloring**. The smallest number of colors needed to color a graph G is called its **chromatic number**, $\chi(G)$. A graph that can be assigned a (proper) k -coloring is **k -colorable**, and it is **k -chromatic** if its chromatic number is exactly k . A subset of vertices assigned to the same color is called a *color class*, every such class forms an independent set. Thus, a k -coloring is the same as a partition of the vertex set into k independent sets, and the terms *k -partite* and *k -colorable* have the same meaning.



This graph can be 3-colored in 12 different ways.

The following table gives the chromatic number for familiar classes of graphs.

	$\gamma(G)$
n	
	$\begin{cases} 3 & \text{for } n \text{ odd} \\ 2 & \text{for } n \text{ even} \end{cases}$
$S_n \quad n > 1$	
$W_n \quad n > 2$	$\begin{cases} 3 & \text{for } n \text{ odd} \\ 4 & \text{for } n \text{ even} \end{cases}$



graph G complete graph K_n cycle graph $C_n, n > 1$

star graph	,	2
wheel graph	,	
	,	2
wheel graph	,	