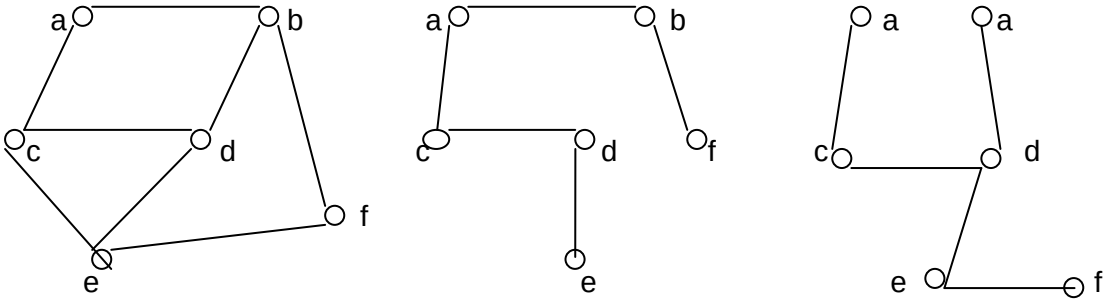
TREE

A graph is said to be tree if it is acyclic.

Spanning tree

A spanning tree T contains all vertices of the graph and has no closed paths.

Example:



(i) connected graph G (ii)spanning tree T_1 (iii)spanning tree T_2

Figure - Graph and spanning trees

Minimum Spanning tree

A spanning tree with least sum of weights among all the spanning trees of a weighted graph is called the minimum spanning tree. A minimum spanning tree is not unique,

Kruskal's algorithm

Let T be the empty spanning tree.

(1) Arrange the set of edges E of the graph in the increasing order of the weights.
(2) Delete the first edge in the set E and add it to the tree T if and only if no closed path is formed in the tree.
(3) Repeat from step 2, if E is not empty. Otherwise stop.
T is the required minimum spanning tree.

Prim's algorithm

Start from any vertex. Build the minimum spanning tree by deleting edges from the edge set E and adding them to the tree T.

(1) Select an arbitrary vertex v ∈ V and add it to V'.

(2) Select an edge (x, y) from E with minimum weight such that one of its end vertices is in V' and other in V – V'. Add this edge to the tree T only if it doesn't cause to create a closed path in T. If more than one edge satisfy this condition then select any of these edges arbitrarily. Also add vertices u and v to V' if already not there.

(3) Repeat from step 2, if V' ≠ V. Otherwise stop.
T is the required minimum spanning tree.

Theorem

The sum of the degrees of all the vertices in a graph is equal to twice the number of edges.

Proof

Let us prove this theorem by induction. Consider a graph G = (V, E).

(1) Let G contains no edge,
i.e. number of edges = 0,

⇒ sum of the degrees of all the vertices = 0,

= 2 x 0,

⇒ theorem is true.

(2) Let G contains one edge. Then there may be either two vertices in G which are the two ends of the edge or the edge may be a loop on one vertex. In both the cases sum of the degrees will be 2.

i.e. number of edges = 1,

⇒ sum of the degrees of all the vertices = 2,

= 2 x 1,

⇒ theorem is true.

(3) Let the theorem is true for an arbitrary graph with number of edges = e,
i.e. sum of the degrees of all the vertices = 2e.

Now add one edge in the graph, then total number of edges = e + 1.

Note that addition of an edge will increase the total degree of the graph by 2

Therefore, the sum of the degrees of all vertices = 2e +2,

= 2 x (e +1),

= 2 x number of edges.

Hence theorem is true.

Theorem

In a graph G, the number of vertices of odd degree is even.

Proof

Consider a graph G = (V, E) such that it contains some vertices of odd and some of even degree. Partition the set of vertices into two categories,

(i) Vertices of odd degree and

(ii) Vertices of even degree.

Then total degree of the graph = sum of degrees of odd degree vertices + sum of degrees of even degree vertices.

Further,

(a) the total degree of the graph is even = 2m, say.

and (b) the sum of the degrees of even degree vertices will be even, naturally = 2n, say.

Therefore,

2m = sum of degrees of odd degree vertices + 2n,

⇒ sum of degrees of odd degree vertices = 2m – 2n,

= 2(m – n),

= even.

⇒ Number of vertices of odd degree will be even; because only the sum of even number of odds can be even.

Hence theorem is proved.

Theorem

The total number of edges in a complete graph with n vertices is n(n – 1)/2.

Proof: Consider a complete graph K_n.

In a complete graph with n vertices, each vertex is connected with other n – 1 vertices.

Therefore degree of each vertex = n – 1,

⇒ total degree = n(n – 1)

⇒ 2 × number of edges in the graph = n(n – 1)

Hence, total number of edges in the graph = n(n – 1)/2.

Theorem

Every cut set in a connected graph G contains at least one edge of every spanning tree of G.

Proof

Consider a graph G and let S be a cut set of it. Also let T be a spanning tree of it.

A spanning tree contains all the vertices of the graph which are connected by the edges. There is no loop or closed path in the tree, and therefore, all the edges are essential to keep the tree a connected graph. Clearly to disconnect the tree, at least one edge has to be deleted. Therefore every cut set in a connected graph G contains at least one edge of every spanning tree of G.

Dijkstra's shortest path algorithm

Consider a connected graph G = (V, E). Let a and z be two vertices. The aim is to find the shortest path between a and z.

Partition the set V of vertices into two sets, P and T, such that P contains a and T = V – P.

(1) Initially P := {a}; mark vertex a visited by enclosing it in a circle.
T := V – {a},
For every vertex r in T let i(r) := w(a, r),
where w(a, r) := ∞, if a and r are not adjacent,
:= weight of the edge (a, r), otherwise.
Call i(r), the index of vertex r.
Also attach path, the sequence of vertices with i(r) whose weight is w(a, r) if w(a, r) ≠ ∞. Leave blank, otherwise. Blank is shown by a dash symbol(–).

(2) Select a vertex x in T that has the minimum index; mark x visited by enclosing it in a circle.
If x = z, stop and exit; shortest path is the
sequence of vertices attached with i(x) and distance is i(x);
otherwise, move to step 3.

(3) P' := P ∪ {x},
T' := T – {x},
for every vertex y adjacent to x in T', modify its index by the formula
i(y) := min{ i(y), i(x) + w(x, y)}.

Also p := p',

T := T'.

Modify the path (or sequence of vertices) attached with each vertex in T according to the new index i(y); i.e. the sequence of the vertices which gives the value i(y).

(4) Go to step 2.

Paths between vertices

Adjacency matrix of a graph can be used to find the paths between each pair of vertices. Let A be the adjacency matrix. Find the powers of A; i.e. A^2, A^3, $\ldots A^n$ etc. Denote an element of adjacency matrix A^k by $a_{k(i,\,j)}$. Then $a_{k(i,\,j)}$ gives the number of paths of length k from vertex v_i to vertex v_j.

Also let $B_r = A + A^2 + A^3 \ \ldots + A^r$. then b_{ij}, element in i^{th} row and j^{th} column gives the number of paths of length r or less from vertex v_i to vertex v_j.

Path matrix

Let G be a graph with n vertices. Then a square matrix P of order $n \times n$ is called the path matrix of the graph G, such that the element

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

= 0, otherwise.

Let A be the adjacency matrix of the graph G with n vertices. Compute A^2 , A^3 , … , A^n.

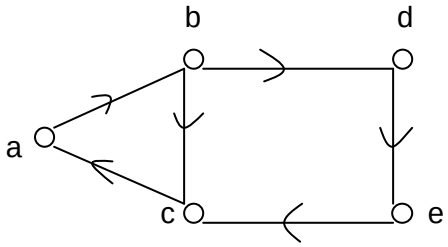
Also find $B_n = A + A^2 + A^3 \ \ldots + A^n$. Form a matrix P by replacing the non zero elements of matrix B_n by 1. This P will be the path matrix of the graph G. If all the elements in the path matrix of a graph are 1, then it is called strongly connected.

Warshall's algorithm

A path matrix is also called **transitive closure** of the graph. Here, the algorithm is written in pseudo Pascal code. Consider a directed graph G =(V, E) and let M be its adjacency matrix. Matrix M is Boolean consisting of elements 0 or 1. Path matrix P is computed by execution of this algorithm.

```
begin
        for i  := 1 to n do
                M[i, i] := 1;
            for i  := 1 to n do
              (
                for j  := 1 to n do
                    (
                      if M[j, i] = 1 then
                          for k := 1 to n do
                                M[j, i] := M[j, i] ∨ M[i, k];
                     )
                    )
            for i  := 1 to n do
               (
                for j  := 1 to n do
                        P[j, i] := M[j, i];
               )
    end;
```

Example



Graph G

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	1	0
c	1	0	0	0	0
d	0	0	0	0	1
e	0	0	1	0	0

Adjacency matrix

P =	a	b	c	d	e
a	1	1	1	1	1
b	1	1	1	1	1
c	1	1	1	1	1
d	1	1	1	1	1
e	1	1	1	1	1

Path matrix

When algorithm is executed completely, the path matrix P is obtained. Since all the elements of P are 1 therefore matrix P is strongly connected.

Tree traversal

A binary tree can only be accessed by its root; i.e. the beginning of the traversal is only possible from its root. Addresses of other vertices in the tree are stored in their preceding vertices.

Preorder tree traversal

Consider a binary tree T with root R.

(1) Process the root R.

(2) Traverse the left sub tree of R in preorder.

(3) Traverse the right sub tree of R in preorder.

Process means performing some operation like reading the data. The algorithm is also read as **root-left-right**

Inorder tree traversal

(4) Traverse the left sub tree of R in inorder.

(5) Process the root R.

(6) Traverse the right sub tree of R in inorder.

It is also read as **left-root-right.**

Postorder tree traversal

(1) Traverse the left sub tree of R in postorder.
(2) Traverse the right sub tree of R in postorder.
(3) Process the root R.

It is read as **left-right-root.**

Notice that all the three algorithms are similar, except the change in the order of execution of the three statements.

Expression trees

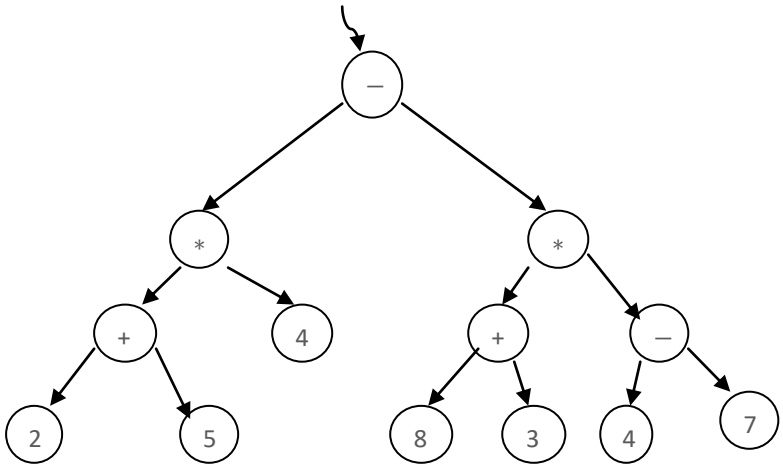
An arithmetic expression can be computed using an expression tree. It is a binary tree that stores numeric data in the leaves of the tree. Operators of the expression are stored in the interior nodes of the tree. Computation is performed in three ways: prefix, infix and postfix that resemble the three traversal procedures given above. For instance, consider expression x = 2 + 3 ∗ 4 − 7. This expression may be evaluated ambiguously resulting in different values if we do not have operator precedence in mind; i.e. it may be (2 + 3) ∗ 4 − 7 or 2 + (3 ∗ 4) − 7 etc. Therefore, expression may be parenthesized to remove the ambiguity. A fully parenthesized expression, like ((2 + 3) ∗ 4) − 7 has no ambiguity and results in one value.

If an expression is fully parenthesized its expression tree will be unique. Construction of an expression tree is easy if the expression is given in infix form.

Let the expression be ((2 + 5) ∗ 4) − ((8 + 3)∗(4 − 7)). This is fully parenthesized and hence is unambiguous. This can be divided into two sub expressions which are on the left and right of the operator '− '.

1. ((2 + 5) ∗ 4) and
2. ((8 + 3)∗(4 − 7))

Operator '−' will form the root of the tree. The first expression will constitute the left sub tree and the second right sub tree.



Postfix notation is also called reverse polish notation (RPN) and it is used in some calculators for arithmetic computation. RPN can easily be computed by using a data structure **stack.** Stack is a data structure in which elements are added and deleted from only one end called top of the stack. The data added last is deleted first. Hence, it is called last in first out (**LIFO**) structure.

Cut point

A vertex in a graph G is called a cut point if its deletion makes the graph disconnected.

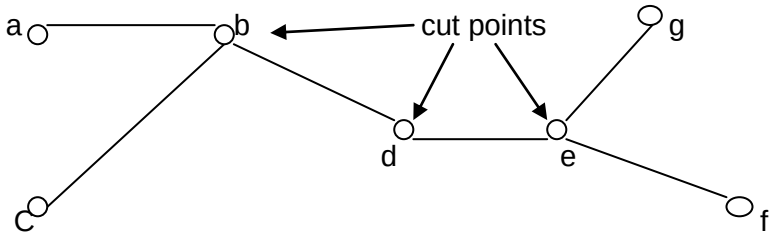
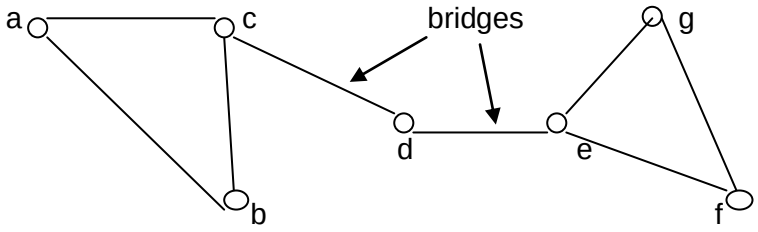


Figure –Cut points

Bridge

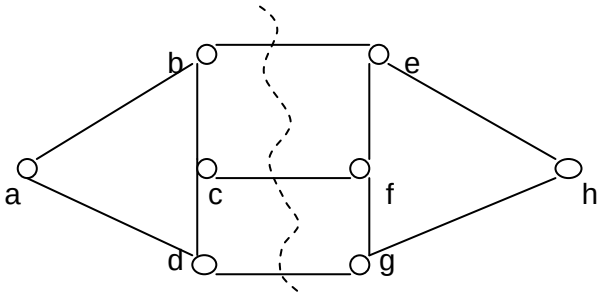
An edge of a graph is called a bridge if its deletion makes the graph disconnected.



Bridges

Edge-cutset

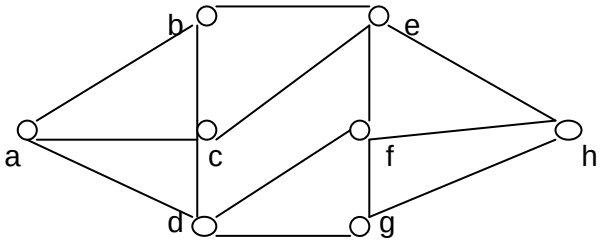
An edge-cutset of a graph is the collection of edges whose removal makes the graph disconnected.



Edge-cutset {(b, e), (c, f), (d, g)}

Vertex-cutset

A vertex-cutset of a graph is the collection of vertices whose removal makes the graph disconnected.



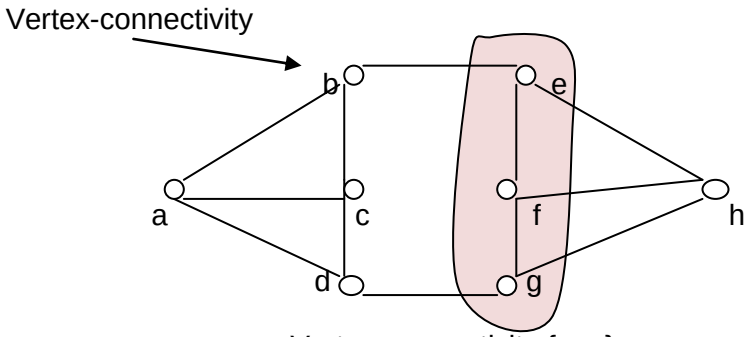
Vertex-cutset {b, c, d}

Partition-cutset

Let G_1 and G_2 be two sets of vertices of graph G such that G_1 and G_2 form a partition of G. Then the collection of all edges of G having one end point in G_1 and the other end point in G_2 is called a partition-cutset. Removal of partition-cutset from the graph makes the graph disconnected.

Vertex-connectivity

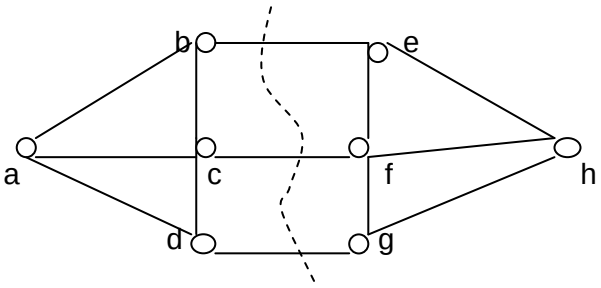
The vertex-connectivity of a connected graph G is the minimum number of vertices whose deletion can either disconnect G or reduce it to a single vertex graph. It is denoted as K_v.



Vertex-connectivity {e, g}

Edge-connectivity

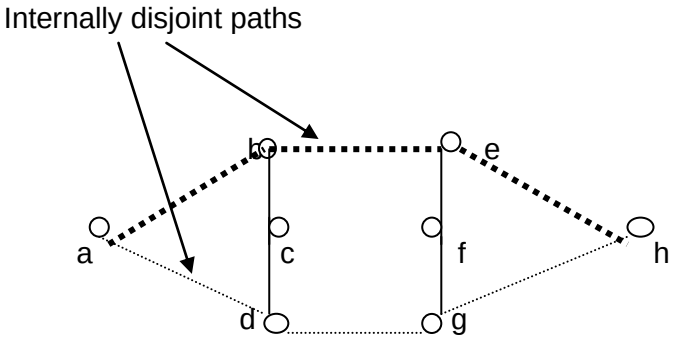
The Edge-connectivity of a connected graph G is the minimum number of edges whose deletion can disconnect G. It is denoted as K_e.



Edge-connectivity $K_e = 3$ with edge-cutset {(b, e), (c, f), (d, g)}

Internally disjoint paths

Let G be a graph. Let a and b be any two of its vertices. Let us call a path of G **a–b path** if a is initial vertex and b is final vertex of the path. A vertex v is called the **internal vertex** of a−b path if it is neither the initial vertex nor the final vertex of the path. A set of **a–b paths** in G is said to be **internally disjoint** if no two paths in the set have an internal vertex in common.



Internally disjoint paths

(viii) k-edge-connected graph

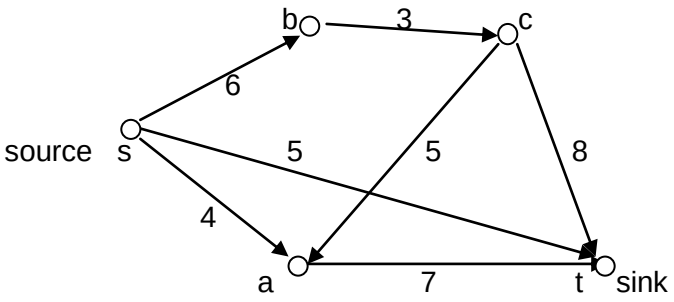
Let G be a graph. Then G is said to be k-edge-connected, if G is connected and $K_e \geq k$.

Menger's theorem

Let G be an undirected graph, and let a and b be two non-adjacent vertices in G. Then, the maximum number of internally-disjoint a-b paths in G equals the minimum number of vertices from V − {a, b} whose deletion separates a and b. Here, V is the set of all vertices of G.

Flow-network

A **flow-network**, G = (V, E) is a directed graph with V the set of vertices (or **nodes**), and E the set of edges (or **arcs**) such that each edge e ∈ E has a **capacity c(e)** ≥ 0. In the flow-network, if a node has only outgoing edges it is called **source**. In contrary, if a node has only incoming edges, it is called **sink**. In graph, source node is designated with **s**, and sink as **t**.



Flow-network

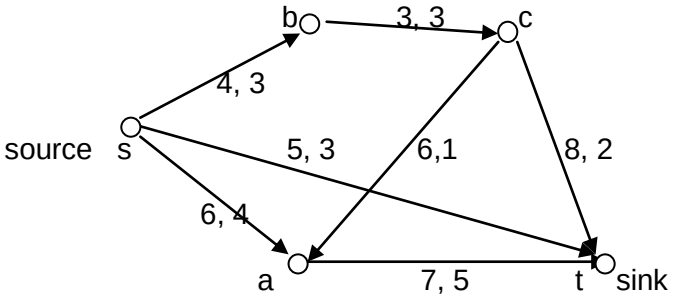
Feasible flow

A **feasible flow** in a flow-network is a function f: E $\longrightarrow \mathbb{R}$; where E is the set of edges, and $\mathbb{R}$, the set of non-negative real numbers. A feasible flow satisfies the following constraints.

(i) Capacity constraint: $f(e) \leq c(e)$**;** where f(e) is the actual assigned flow value to an edge, and c(e), the capacity of edge e.

(ii) Conservation constraint: for every vertex v, $\sum$ In(v) = $\sum$ Out(v); where, In(v) is the incoming flow, and Out(v), outgoing flow of node v.

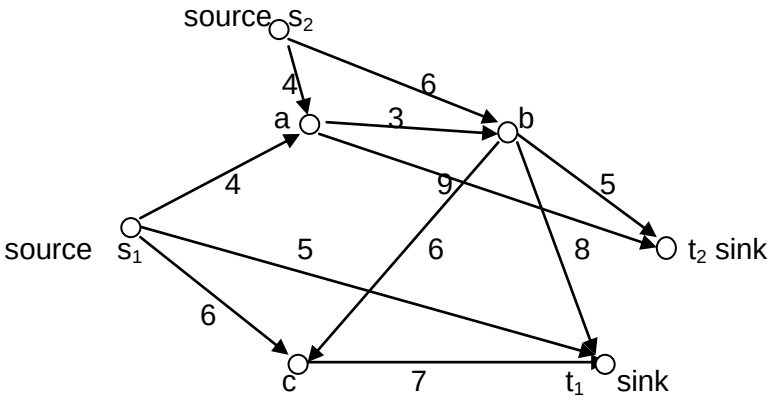
Each edge of the flow-network in a feasible flow has two labels: capacity and actual flow. First digit shows the capacity of the edge while second, actual flow through it.



A feasible flow in a flow-network

Multiple Sources/Sinks

A flow-network may contain more than one source, and more than one sink nodes.



Multiple Sources/Sinks

Cut sets in flow-networks

Let F be a flow-network. Partition the set of vertices V into two sets V_s and V_t such that source $s \in V_s$, and sink $t \in V_t$. Since, V_s and V_t form a partition of V, therefore, it is obvious that $V_s \cup V_t = V$, and $V_s \cap V_t = \emptyset$.

Set of all edges, which are directed from vertex set Vs to vertex set V_t is called an s-t cut and is denoted as $< V_s, V_t >$. A **minimum s-t cut** is a cut with minimum capacity.

Relation between flows and cuts

There is a direct relationship between a flow and a cut. In fact, the value of any flow equals the total flow across the edges of the cutset $< \{ s \}, V-\{s\} >$. Here, V is the set of vertices, and $V_s = \{s\}$ and $V_t = V - \{s\}$. Let **Val(f)** denotes the **net flow** leaving the source s, then

$$\text{Val(f)} = \sum_{e \in \text{Out}(s)} f(e) - \sum_{e \in \text{In}(s)} f(e).$$

In other words,

$$\text{Val(f)} = \sum_{e \in <\{s\},\ V-\{s\}>} f(e) - \sum_{e \in <V-\{s\},\ \{s\}>} f(e).$$

Maximum flow problem

The maximum flow in an s-t network can be found by augmenting the flow iteratively until no further augmentation is possible. It may be seen that an assignment of a maximum flow is not unique. Let f be a flow in a network G (V, E) , and let there be a directed path

$p = s\text{-}e_1\text{-}v_1\text{-}e_2\text{-}v_2\text{-}\ldots\text{-}e_k\text{-}t$

such that $f(e_i) < c(e_i)$ for i = 1 to k, where $s, v_1, v_2, \ldots, t \in V$, and $e_1, e_2, \ldots, e_k \in E$.

Now considering the capacity on each edge, the flow can be augmented by as much as $\Delta_p = c(e_i) - f(e_i)$. However, to maintain the conservation of flow at each node v_i, increase on all the edges of the path must be equal.

Thus, the largest possible value of Δ_p = minimum $\{c(e_i) - f(e_i)\}$.

f-augmenting path

A **semi s-t path** is a sequence of vertices and edges $p = s\text{-}e_1\text{-}v_1\text{-}e_2\text{-}v_2\text{-}\ldots\text{-}e_k\text{-}t$ such that each vertex except t, is followed by an edge, which may be either a **forward edge** or a **backward edge**. An edge e_i is called a **forward edge** if it is directed from vertex v_{i-1} to vertex v_i, and edge e_i is called **backward edge** if it is directed from v_i to v_{i-1}. An **f-augmenting path** is a semi s-t path such that the flow on each forward edge can be increased, and the flow on each backward edge can be decreased.

For each edge e on an f-augmenting path, the quantity Δ_e is called the **slack** on edge e, and is given by,

$$\Delta_e = \begin{cases} c(e) - f(e), & \text{if e is a forward edge,} \\ f(e), & \text{if e is a backward edge .} \end{cases}$$

Max-flow min-cut theorem

The theorem states that for a given network, the value of a maximum flow is equal to the capacity of a minimum cut.

Algorithm for finding maximum flow

```
begin
 for each edge e in a network G
 f(e) :=0;
 while there exists an f-augmenting path in a network
  begin
  find an f-augmenting path p;
  let Δp = minimum{Δe}, where e∈ p;
  for each edge e ∈ p
  begin
   if e is a forward edge then
   f(e) := f(e) + Δp
   else
       f(e) := f(e) − Δp;
    end;
   end
   end;
```

Matching in a graph

A **matching** in a graph G is a set of edges, which have no endpoints in common. It is also called a **matching set** and is denoted by M. A **maximum matching** is a matching with maximum number of edges. A matching is called **maximal,** if it is not a proper subset of any other matching. A **perfect matching** is a matching which matches all vertices of the graph; that is, every vertex of the graph is the end vertex of an edge of the matching. Note that if a graph has odd number of vertices, then it cannot have a perfect matching. Every perfect matching is maximum, and thus maximal. A **near-perfect matching** is one in which exactly one vertex is unmatched. This may occur when a graph has an odd number of vertices, and the matching defined is maximum.

Matching in a bipartite graph

We know that in a bipartite graph a vertex bipartition exists. Let G be a bipartite graph such that $< X, Y >$ be a bipartition, then the set of edges having one end in X and other in Y without common end points is a matching.

Covering in a graph

Vertex-cover

Let G(V, E) be a graph. Let C be a subset of vertex set V. Then the subset C is called a **vertex-cover** of graph G if every edge of G is incident on at least one vertex in C.

Obviously, the set of all vertices V is a vertex-cover. A vertex-cover is called **minimum vertex-cover** if it has least number of vertices. A minimum vertex-cover of a graph can be obtained by finding all possible vertex-covers exhaustively, and then selecting the minimum one. However, the minimum may not be unique.

Note

Let G(V, E) be a graph. Let M be a matching in G, and C be a vertex-cover of it.

- Then $|M| \leq |C|$.
- If $|M| = |C|$, then M is a maximum matching and C a minimum vertex-cover. The converse may not be true in general. However, it does hold if G is bipartite.
- If G is bipartite then the number of edges in a maximum matching in G is equal to the number of vertices in a minimum vertex-cover of G.

Edge-cover

An edge-cover of a graph G(V, E) is a subset C of E, such that each vertex of G is incident with at least one edge in C. A **minimum edge-covering** is an edge-covering of least size. A perfect matching is always a minimum edge-covering. The set of all edges in a connected graph is an edge-cover. A minimum edge-cover of a graph can be obtained by finding all possible edge-covers exhaustively, and then selecting the minimum one.

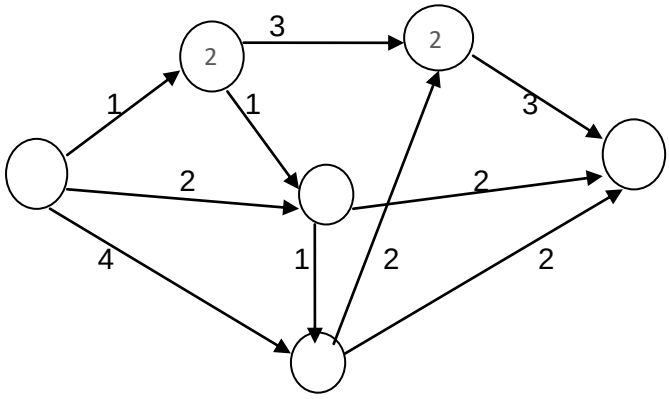
PERT

PERT means program evaluation and review technique. It is a statistical tool used in project management, which was designed to analyze and represent the tasks involved in completing a given task/project.

It was developed by the United States Navy in 1958, it is commonly used in conjunction with the critical path method (CPM).

PERT is a method of analyzing the jobs involved in completing a given project, especially the time needed to complete each job, and to identify the minimum time needed to complete the total project.

It incorporates uncertainty by making it possible to schedule a project. It is more of an event-oriented technique rather than start- and completion-oriented, and is used more in these projects where time is the major factor rather than cost. PERT is a management tool, which is implemented by arrow/directed edge and node/vertex diagram of activities and events.



PERT graph