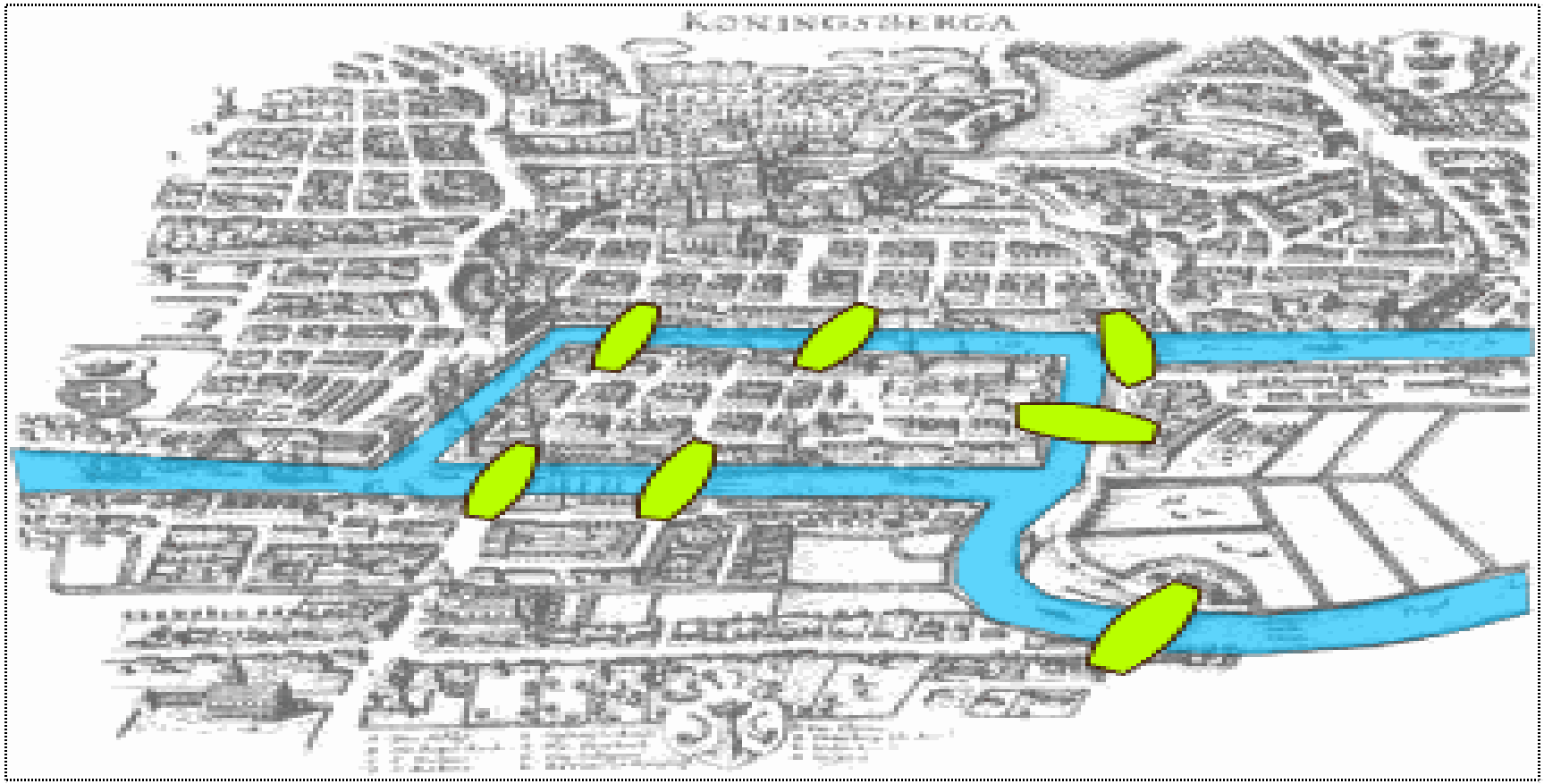
GRAPH : another Non-Linear Data Structure

****

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.



The paper written by Leonhard Euler on the Seven Bridges of Königsberg and published in 1736 is regarded as the first paper in the history of graph theory. This paper, as well as the one written by Vandermonde on the knight problem, carried on with the analysis situs initiated by Leibniz.

After a 100 years of Euler's paper on the bridges of Königsberg, Cayley studied a particular class of graphs, the trees.

Euler's work was presented to the St. Petersburg Academy on 26 August 1735, and published as Solutio problematis ad geometriam situs pertinentis (The solution of a problem relating to the geometry of position) in the journal Commentarii academiae scientiarum Petropolitanae in 1741.

A graph data structure consists of a finite set of vertices or nodes or points, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges, arcs, or lines for an undirected graph and as arrows, directed edges, directed arcs, or directed lines for a directed graph.

Operations

The basic operations provided by a graph data structure G usually include:

- adjacent(G,x,y): tests whether there is an edge from the vertex x to the vertex y;
- neighbors(G,x): lists all vertices y such that there is an edge from the vertex x to the vertex y;
- add_vertex(G,x): adds the vertex x, if it is not there;
- remove_vertex(G,x): removes the vertex x, if it is there;
- add_edge(G,x,y): adds the edge from the vertex x to the vertex y, if it is not there;
- remove_edge(G,x,y): removes the edge from the vertex x to the vertex y, if it is there;
- get_vertex_value(G,x): returns the value associated with the vertex x;
- set_vertex_value(G,x,v): sets the value associated with the vertex x to v.

Structures that associate values to the edges usually also provide:

- get_edge_value(G,x,y): returns the value associated with the edge (x,y);
- set_edge_value(G,x,y,v): sets the value associated with the edge (x,y) to v.

Representations

Different data structures for the representation of graphs are ...

Adjacency List

- Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices.
- This data structure allows the storage of additional data on the vertices.
- Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

Adjacency Matrix

- A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices.
- Data on edges and vertices must be stored externally.
- Only the cost for one edge can be stored between each pair of vertices.

Incidence Matrix

- A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges.
- The entries indicate whether the vertex at a row is incident to the edge at a column.

Adjacency lists are generally preferred because they efficiently represent **sparse** graphs.

An **adjacency matrix** is preferred if the graph is **dense**, $|E|$ is close to $|V|^2$, or if one **must be able to quickly look up if there is an edge connecting two vertices**.

Tree is special form of graph,

minimally connected graph and having only one path between any two vertices. Tree is a special case of graph having **no loops**, **no circuits** and **no self-loops**.

Types of Graphs

A simple graph, also called a strict graph (Tutte 1998), is an unweighted, undirected graph containing no graph loops or multiple edges.

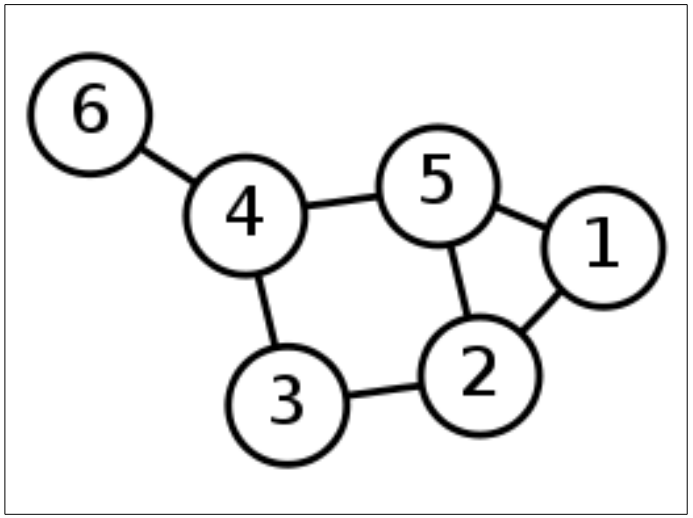
A simple graph may be either connected or disconnected.

There are many different types of graphs ...

- connected and disconnected graphs
- bipartite graphs
- weighted graphs [negative values as profits, otherwise costs]
- labeled graphs
- directed and undirected graphs
- cyclic or acyclic graphs
- simple graphs

Adjacency:

If (u,v) is in the edge set we say u is adjacent to v (written as u ~ v).



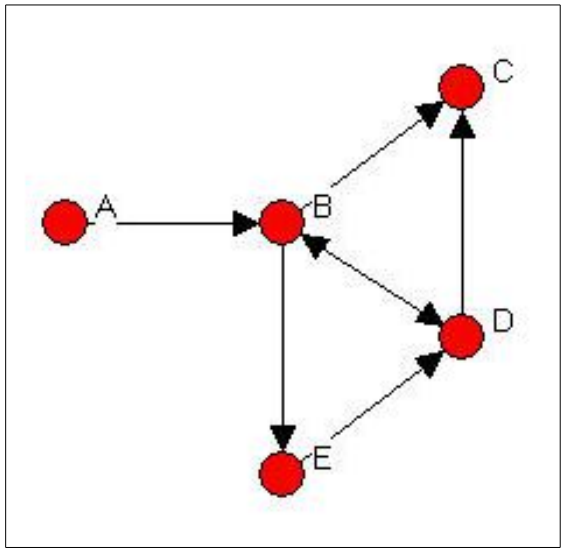
The shown graph has the following parts...

- The underlying set for the Verticies set is the integers [labels]
- The Vertices set = {1,2,3,4,5,6}
- The Edge set = {(6,4),(4,5),(4,3),(3,2),(5,2),(2,1),(5,1)}

Undirected Graphs

- In an undirected graph, the order of the vertices in the pairs in the Edge set doesn't matter.
- For the sample graph,
  Edge set, E := {(4,6),(4,5),(3,4),(3,2),(2,5)),(1,2)),(1,5)}.
- Undirected graphs usually are drawn with straight lines between the vertices.
- The adjacency relation is symetric in an undirected graph [u ~ v = v ~ u].

Directed Graphs



- In a directed graph the order of the vertices in the pairs in the edge set matters.
- Thus u is adjacent to v only if the pair (u,v) is in the Edge set.
- For directed graphs we usually use arrows for the arcs between vertices.
- An arrow from u to v is drawn only if (u,v) is in the Edge set.

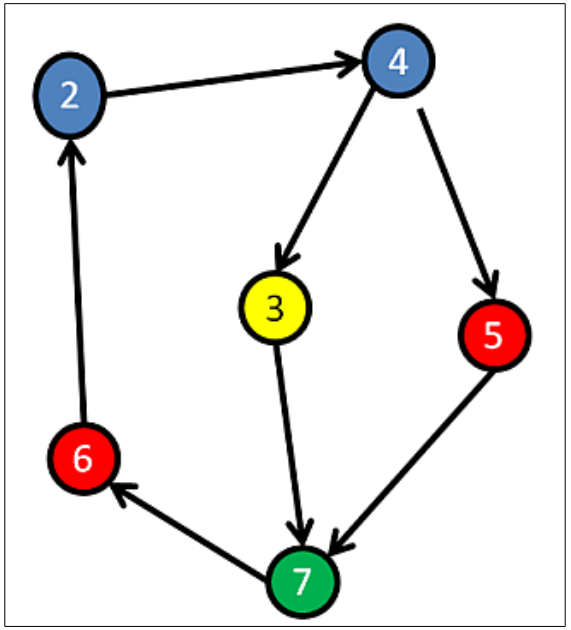
The shown directed graph has the following parts...

- The underlying set for the Verticies set is capital letters.
- The Vertices set = {A,B,C,D,E}
- The Edge set = {<A,B>,<B,C>,<D,C>,<B,D>,<D,B>,<E,D>,<B,E>}

  Note that both <B,D> and <D,B> are in the Edge set, so the arc between B and D is an arrow in both directions.

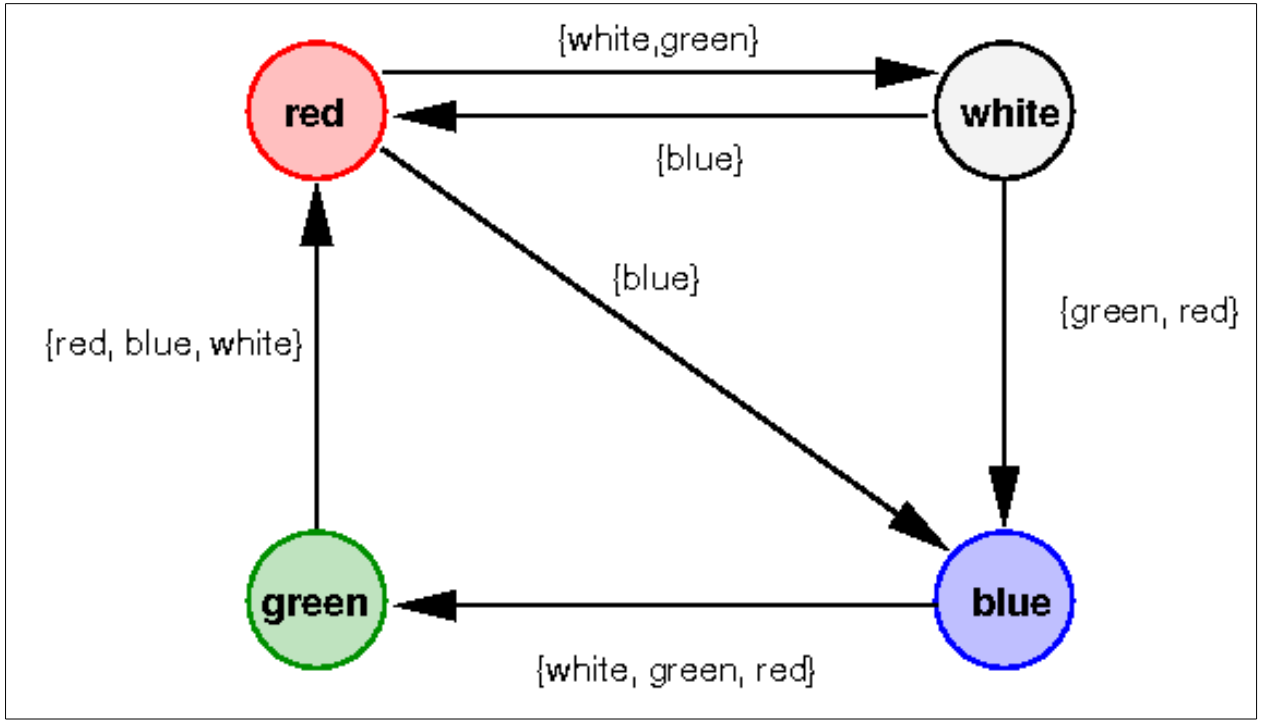
Cyclic Graphs

- A cyclic graph is a directed graph with at least one cycle.
- A cycle is a path along the directed edges from a vertex to itself.
- The vertex labeled graph has several cycles.
  One of them is 2 » 4 » 5 » 7 » 6 » 2



Edge Labeled Graphs

- A Edge labeled graph is a graph where the edges are associated with labels.



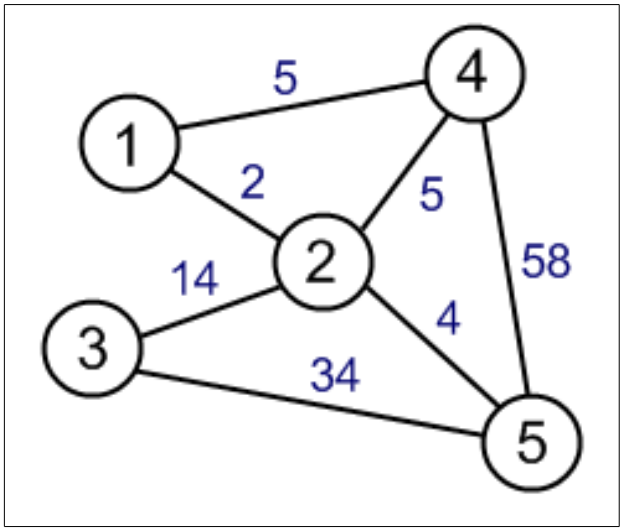
- The Edge set is a set of triples. Thus if (u,v,X) is in the edge set, then there is an edge from u to v with label X.
- Edge labeled graphs are usually drawn with the labels drawn adjacent to the arcs specifying the edges.

The edge-labeled graph above has the following parts...

- The underlying set for the the Vertices set is Color.
- The underlying set for the edge labels is sets of Color.
- The Vertices set = {Red,Green,Blue,White}
- The Edge set =
  {(red,white,{white,green}),(white,red,{blue}),(white,blue,{green,red}),
  (red,blue,{blue}),(green,red,{red,blue,white}),(blue,green,{white,green,red})}

Weighted Graphs

- A weighted graph is an edge labeled graph where the labels can be operated on by the usual arithmetic operators.
- Usually costs [or profits] are integers or floats.



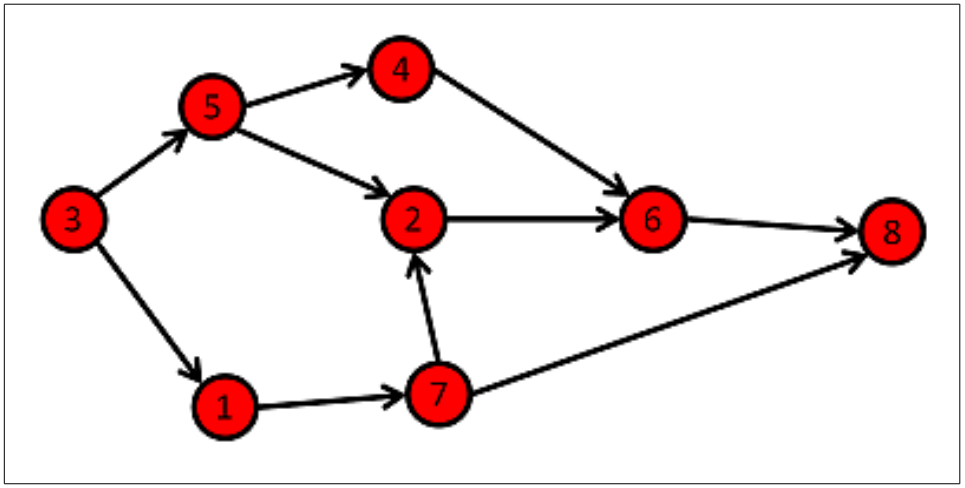
The weighted graph shown hasthe following parts...

- The underlying set for the Vertices set is Integer.
- The underlying set for the weights is Integer.
- The Vertices set = {1,2,3,4,5}
- The Edge set,

  E := {(1,4,5),(4,5,58),(3,5,34),(2,4,5),(2,5,4),(3,2,14),(1,2,2)}

Directed Acyclic Graphs

- A DAG is a directed graph without cycles.



The DAG shown has the following parts...

- The underlying set for the the Vertices set is Integer.
- The Vertices set = {1,2,3,4,5,6,7,8}
- The Edge set = {<1,7>,<2,6>,<3,1>,<3,5>,<4,6>,<5,4>,<5,2>,<6,8>,<7,2>,<7,8>}

Disconnected Graphs

- Vertices in a graph do not need to be connected to other vertices.
- It is legal for a graph to have disconnected components, and even lone vertices without a single connection.

A graph with following chacteristics is disconnected...

- The underlying set for the the Vertices set is Integer.
- The Vertices set = {1,2,3,4,5,6,7,8}
- The Edge set = {(1,7) ,(3,1),(3,8) ,(4,6) ,(6,5)}

In directed graphs ..

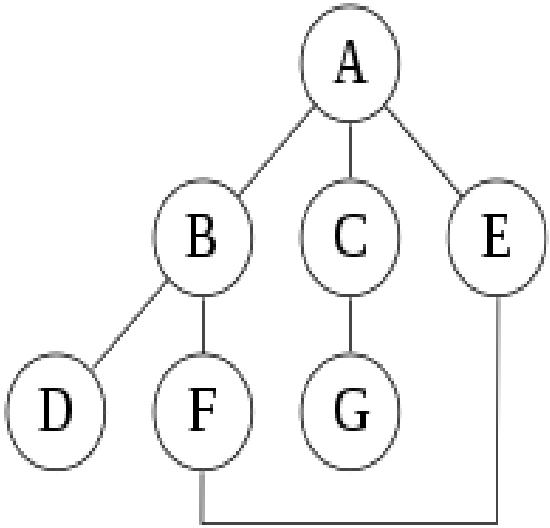
- Vertices with only in-arrows are called sinks.
- Vertices with only out-arrows are called sources.

GRAPH TRAVERSALS

Depth First Search (DFS)

This traversing method uses the stack for storing the visited vertices. DFS is the edge based method and works in the recursive fashion where the vertices are explored along a path (edge). The exploration of a node is suspended as soon as another unexplored node is found and the deepest unexplored nodes are traversed at foremost. DFS traverse/visit each vertex exactly once and each edge is inspected exactly twice.

For the graph shown a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.



```
Procedure DFS(U)
    Visit(U)
    For each V such that (U, V) is an edge do
        If V was not visited yet Then
            DFS(V)
```

```
Procedure DFS(G, SRC, VISITED)
        G[|V|][|V|] is an adjacency matrix for G(V,E).
        SRC is starting vertex for DFS.
        VISITED stores vertices that are visited.
        |V| indicates number of vertices in G(V,E).
        VKNT is the vertex index [ 0 .. |V|-1 ]
1.  Initialize counter
        VKNT := 0
2.  Visit the vertex and print it
        VISITED[SRC] := TRUE
        Write(SRC)
3.  Iterate over all non-visited vertices recursively
        While VKNT < |V|
            If G[SRC][VKNT]=1 AND VISITED[VKNT]=FALSE
                Call DFS(G,VKNT,VISITED)
            VKNT:= VKNT+1
```

DFS Applications:

1. Cycle detection in a graph
2. Solving a maze [DFS explores every possible path before backtracking]
3. Checking if a graph is bipartite or not
4. Finding out strongly connected components of a graph.
5. Longest path between two nodes in a graph

Breadth First Search (BFS)

This traversing method used in graphs. It uses a queue for storing the visited vertices. In this method the emphasize is on the vertices of the graph, one vertex is selected at first then it is visited and marked. The vertices adjacent to the visited vertex are then visited and stored in the queue sequentially.

Similarly, the stored vertices are then treated one by one, and their adjacent vertices are visited.

A node is fully explored before visiting any other node in the graph, in other words, it traverses shallowest unexplored nodes first.

BFS Applications:

- Shortest Path and Minimum Spanning Tree for unweighted graph
- In P2P (Peer-to-Peer) Networks [BitTorrent] to find all neighbor nodes.
- Crawlers in Search Engines [source page to target page traversal]
- Social Networking Websites [popularity and prestige of a person]
- GPS Navigation Systems [explore all neighboring locations]
- Broadcasting in Network [reaching to all nodes, IP Address: 255.255.255.255]
- In Garbage Collection [copying garbage collection]
- Cycle detection in undirected graph [also DFS - directed graph]
- Ford–Fulkerson algorithm [maximum flow, either DFS|BFS]
- To test if a graph is Bipartite [either DFS|BFS]
- To find a path between two vertices.
- Finding all nodes within one connected component [DFS|BFS]
- Extended into Prim's MST and Dijkstra's SSSP algorithm

```
Algorithm Procedure BFS(U)
    Visit(U)
    Queue := U                      [enqueue]
    While Queue is not empty Do
        U := Queue                  [dequeue]
        For each V such that (U,V) is an edge
                    and V has not been visited yet Do
                Visit(V)
                Queue := V          [enqueue]

Procedure BFS(G, SRC, VISITED)
    G[|V|][|V|] is an adjacency matrix. SRC is starting vertex for BFS.
    VISITED stores vertices that are visited.
    |V| indicates number of vertices in G(V,E).
    Q is linear queue using an array of MX elements [REAR, FRONT].
    VKNT is the vertex index [ 0 .. |V|-1 ]
```

```
1.  Initialize queue pointers
        FRONT:= -1, REAR:= -1
2.  Visit the vertex and print it
        VISITED[SRC] := TRUE
        Write(SRC)
3.  Enqueue the (source) vertex
        REAR:= REAR+1, FRONT:= FRONT+1
        Q[REAR]:= SRC
4.  Iterate until queue is not Empty
        While FRONT <= REAR Repeat Step 05 thru 07
5.  Remove Vertex from Q [dequeue] & increment FRONT
        SRC:= Q[FRONT]
        FRONT:= FRONT+1
6.  Initialize the Counter
        VKNT:= 0
```

7. Iterate over all non-visited vertices

```
While VKNT < |V|
    If G[SRC][VKNT]=1 AND VISITED[VKNT]=FALSE
        VISITED[VKNT] := TRUE
        Write(VKNT)
        //Enqueue the vertex
        REAR:= REAR+1
        Q[REAR]:= VKNT
    VKNT:= VKNT+1
```

BFS Versus DFS : Key Differences

1. BFS is vertex-based algorithm while DFS is an edge-based algorithm.
2. Queue data structure is used in BFS, while DFS uses stack or recursion.
3. Memory space is efficiently utilized in DFS while space utilization in BFS is not effective.
4. BFS is optimal algorithm while DFS is not optimal.
5. DFS constructs narrow and long trees. As against, BFS constructs wide and short tree.

When to use BFS|DFS??

- use BFS - when prime motive is to to find the shortest path from a certain source node to a certain destination.
- use DFS - when prime motive is to exhaust all possibilities, and check which one is the best/count the number of all possible ways.
- use either BFS or DFS - when checking the connectedness between two nodes on a given graph.

DIJKSTRA'S METHOD [Edsger W. Dijkstra, 1956] [Leyzorek 1957]

The Single Source Shortest Path [SSSP] Algorithm

It is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$(where $|V|$ is the number of nodes).

In the following algorithm, the code U←vertex in Q with Min_Dist[U], searches for the vertex U in the vertex set Q that has the least dist[u] value. Length(U,V) returns the length of the edge joining (i.e. the distance between) the two neighbor-nodes U and V. The variable ALT is the length of the path from the root node to the neighbor node V if it were to go through U. If this path is shorter than the current shortest path recorded for V, that current path is replaced with this ALT path. The prev array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

Non-Negative Edges only. Greedy Approach.

```
Function Dijkstra(G, SRC):
    Create Vertex Set Q
    For each vertex V in G:      // Initialization
        DIST[V] ← INFINITY       // Unknown distance from SRC to V
        PREV[V] ← UNDEFINED      // Previous vertex in optimal path from SRC
        Add V to Q               // All vertices initially in Q (unvisited)
    DIST[SRC] ← 0                // Distance from SRC to SRC

    While Q is not empty:
        U ← vertex in Q with Min_Dist[U]   // Select Vertex with least distance
        Remove U from Q
        For each neighbor V of U:          // where V is still in Q
            ALT ← DIST[u] + Length(U,V)
            If ALT < DIST[v]:              // A shorter path to V exists
                DIST[V] ← ALT
                PREV[V] ← U
    Return DIST[], PREV[]
```

WARSHALL-FLOYD METHOD

[All Sources Shortest Path | Reachability Matrix | Path Matrix]

Given a directed graph G(V,E), a vertex J is reachable from a vertex I if there exists a path from vertex I to J. Floyd-Warshall algorithm is used to compute the reachability matrix (transitive closure) of a graph.

The Warshall-Floyd algorithm is uesd to find shortest paths in a weighted graph with both positive and negative edge weights but with “no negative weight cycle”.

A single execution of the algorithm will find the lengths of shortest paths between all pairs of vertices and is an example of dynamic programming.

Pre-Conditions:

The adjacency matrix representation of the graph G is given. Additionally two arrays are used..

1. DIST[][] is a |V|x|V| array of minimum distance. Initialized to INFINITY.
2. NEXT[][] is a |V|x|V| array of vertex indices. Initialized to NULL.

```
Procedure Flyod_Warshall(G)
    For Each Edge (U,V):
        DIST[U][V] := G[U][V],  NEXT[U][V] := V     // Set DIST[U][U] := 0
    For K := 1 to |V|
        For I := 1 to |V|
            For J := 1 to |V|
                If DIST[I][J] > ( DIST[I][K] + DIST[K][J] ) Then
                    DIST[I][J] := DIST[I][K] + DIST[K][J]
                    NEXT[I][J] := NEXT[I][K]
Procedure Path(U,V)
    If NEXT[U][V] = NULL
        Return []
    PATH :=  [U]
    While U <> V
        U := NEXT[U][V]
        PATH.Append(U)
    Return Path
```

SPANNING TREES

A spanning tree of a graph is a tree that has all the vertices of the graph connected by some edges.

A graph can have one or more number of spanning trees.

A spanning tree does not have cycles and it cannot be disconnected.

If the graph has N vertices then the spanning tree will have N-1 edges.

A complete undirected graph of N vertices can have maximum N^{N-2} spanning trees.

Thus, a spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G, with minimum possible number of edges.

A minimum spanning tree (MST) is a spanning tree that has the minimum weight than all other spanning trees of the graph.

The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

General Properties of Spanning Tree

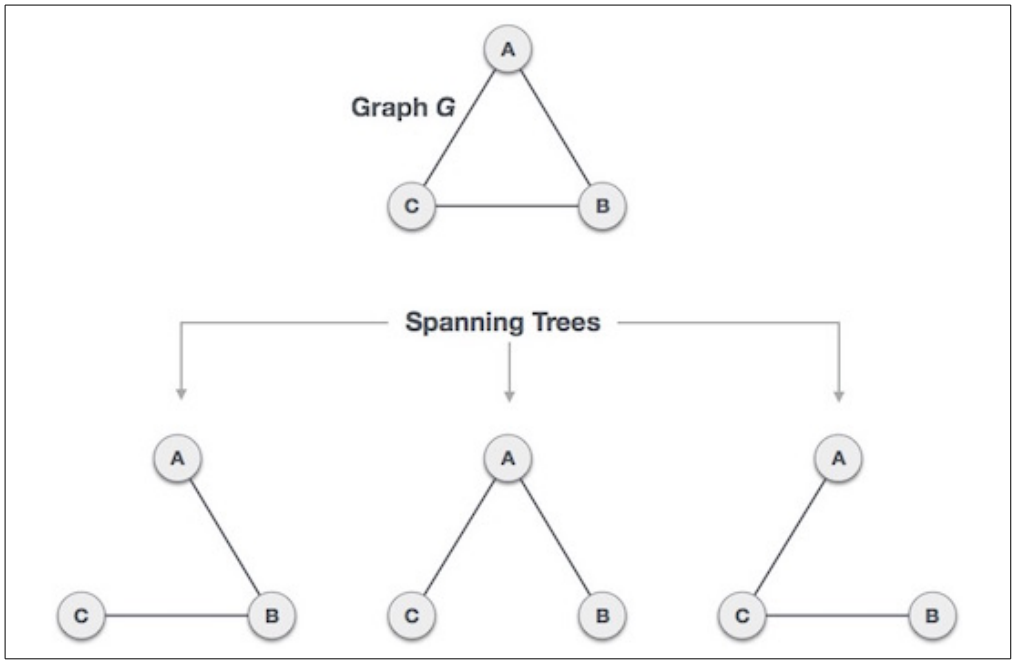
- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have same number of edges & vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

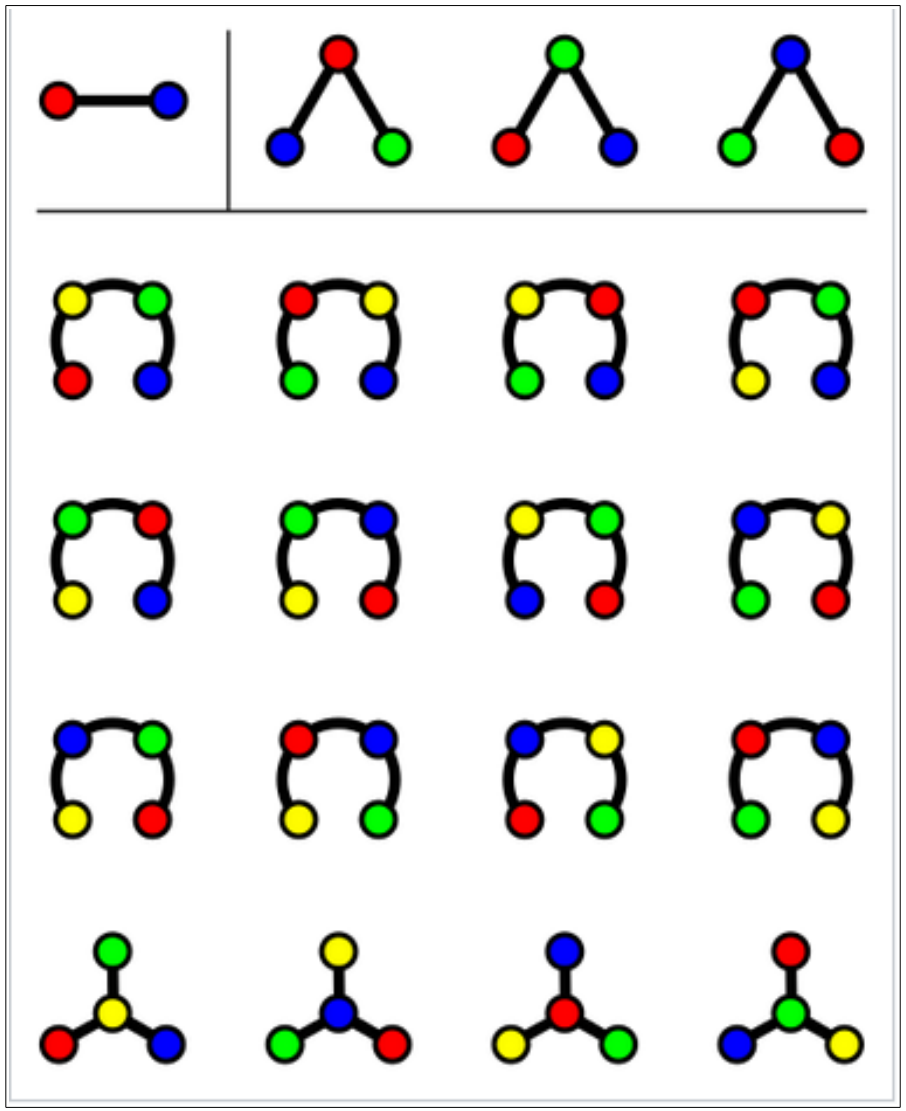
Mathematical Properties of Spanning Tree

- For N vertex graph, a spanning tree has N-1 edges.
- Removing E-N+1 edges from a complete graph, will construct a spanning tree.
- A complete N vertex graph can have maximum N^{N-2} number of spanning trees.
- Spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Spanning Trees

[For G(V,E) with 2, 3, 4]





PRIM'S MST [Vojtěch Jarník 1930, Robert Prim 1957, Edsger W. Dijkstra 1959]

[The Minimum Cost Spanning Tree]

This method is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

The algorithm builds a tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

Majorly Prim's algorithm only finds minimum spanning trees in connected graphs. However, to find the minimum spanning forest, the Prim's algorithm can be separately run for each connected component of the graph.

```
PRIM(G):
    Create Vertex Set Q
    For each vertex V in G:
        DIST[V] ← INFINITY    // Unknown distance from SRC to V
        PREV[V] ← UNDEFINED   // Previous vertex in MST for this edge
        VISITED[V] ← FALSE    // All vertices initially unvisited
    DIST[0] ← 0               // Distance from SRC to SRC
    KNT := 0
    While KNT < |V|-1  Step 1 Do
        U := Min_Dist[U]
        VISITED[U] := TRUE
        VKNT := 0
        While VKNT < |V| Step 1 Do
            If G[U][VKNT]=1 AND VISITED[VKNT]=FALSE
               AND G[U][VKNT] < DIST[V] Then
                DIST[VKNT] ← G[U][VKNT]
                PREV[VKNT] ← U
    Return DIST[], PREV[]
```

KRUSKAL'S MST [Joseph Kruskal 1956]

Theis method finds the minimum cost spanning tree using the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

Method:

1. Remove all loops ad parallel edges.
2. Arrange all edges in their increasing order of weight.
3. Till |V|-1 edges in the Tree
   - Choose a least weight edge
   - Check if it does not introduce cycle in the existing tree
   - Add the edge to the tree otherwise discard the edge

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

Kruskal's algorithm is a MST algorithm which finds an edge of the least possible weight that connects any two trees in the forest.

It is a greedy algorithm as it finds a MST for a connected weighted graph adding increasing cost arcs at each step.

Thus, it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest.

```
KRUSKAL(G):
1   A = ∅
2   For Each V ∈ G.V:
3       MAKE-SET(V)
4   For Each (U,V) in G.E ordered by Weight(U,V), increasing:
5       If FIND-SET(U) ≠ FIND-SET(V):
6           A = A ∪ {(U,V)}
7           UNION(U,V)
8   return A
```

PRIM'S Method versus KRUSKAL'S Method

- Use Prim's algorithm when the graph is dense and Kruskal's algorithm when the graph is sparse.
- Kruskal can have better performance if the edges can be sorted in linear time, or are already sorted. Prim's better if the #edges outweighs the #vertices.
- When terminated in middle Prim's algorithm always generates connected tree, but Kruskal may result in a disconnected tree or forest.
- Kruskal time complexity worst case is O(E log E) [need sorting of edges]. Prim time complexity worst case is O(E log V) with priority queue, O(E+V log V) with Fibonacci Heap.
- In Kruskal Algorithm the starting point is a non-cyclic or non-closed graph. [You need to remove all parallel and self edges]
- Kruskal builds a minimum spanning tree by adding one edge at a time. Prims builds a mimimum spanning tree by adding one vertex at a time.