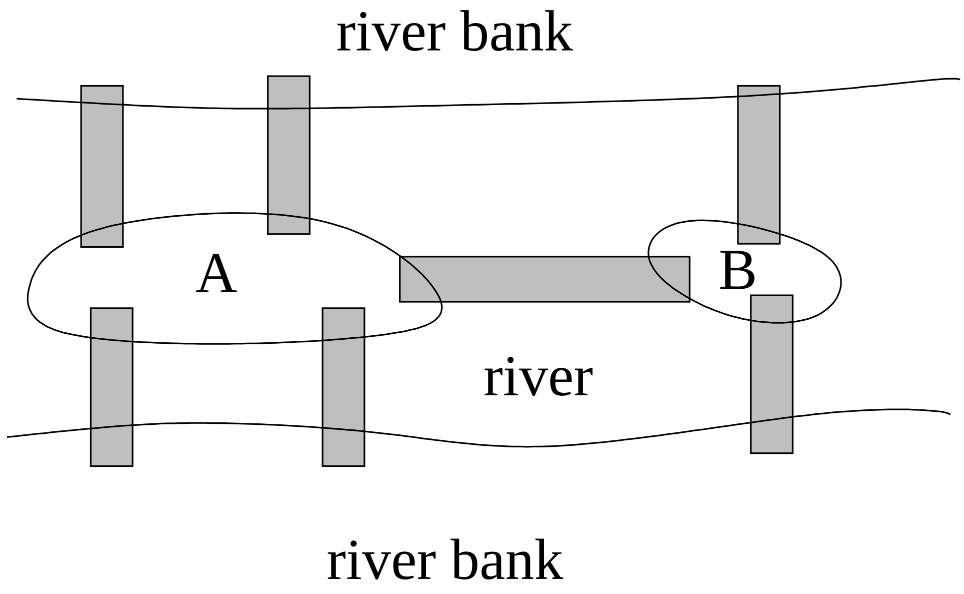
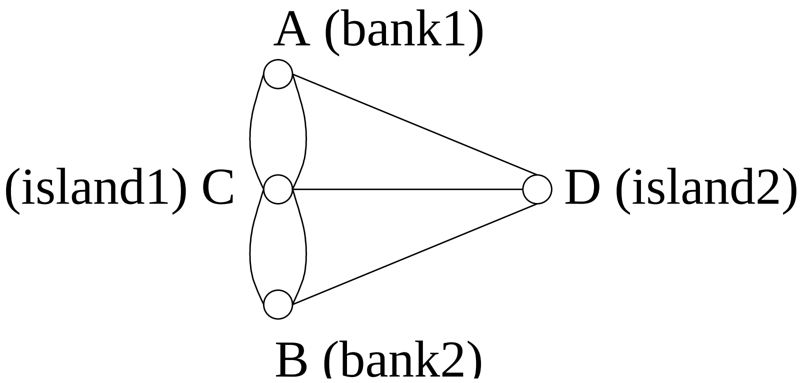
Graphs

- Leonhard Euler is the pioneer – first paper in 1736.
- Euler's problem about Konigsberg's bridges.



Graphs

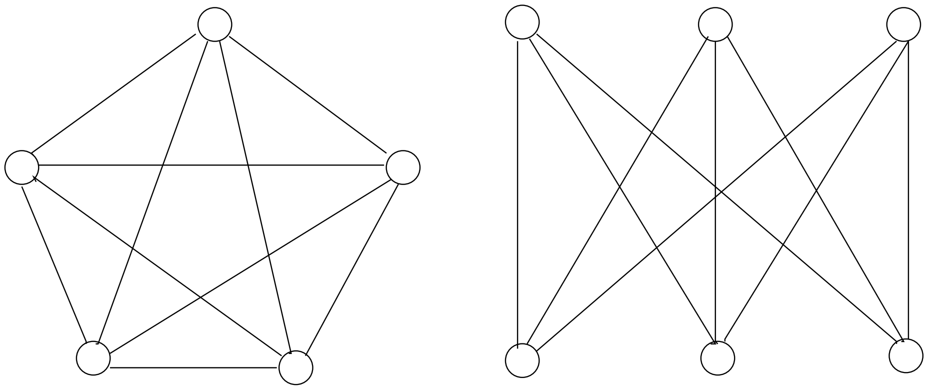
- Can be modeled in terms of a graph.
- Bridges are edges. Each separate continuous landmass represented by a different node.
- Each river bank is a different node, each island is a distinct node
- So we have 4 nodes and 7 edges as shown in th graph below.



Graphs

- Used to represent all kind of relationships between all pairs of objects.
- For example connectivity relationships among the nodes of internet can be represented.
- Trees can be graphs too. Connectivity relationship of parent-child exists among nodes.
- An abstract description of a graph: a non empty set of vertices V and a set of edges E. $E \subseteq V \times V$ can also be empty.

Example



Terminology

- Undirected graphs
- Parallel edges, end vertices of an edge, self-loops
- Degree of vertices.
- Directed graphs, directed edges, initial and terminal vertices of a directed edge.
- Paths, cycles, lengths of paths and cycle.

Terminology contd.

- Eccentricity of a vertex is the maximum distance from any other vertex of the graph, i.e., $e(v) = \max\{dist(v, w)|w \in V\}$
- Radius is minimum eccentricity among vertices of G, i.e., $radius(G) = \min\{e(v)|v \in V\}$.
- Diameter is the maximum eccentricity among vertices in G
- Center of G consists of set of veritces with minimum eccentricity, i.e., $center(G) = \{v \in V|e(v) = radius(G)\}$.
- Girth of G is the length of a shortest cycle in G.

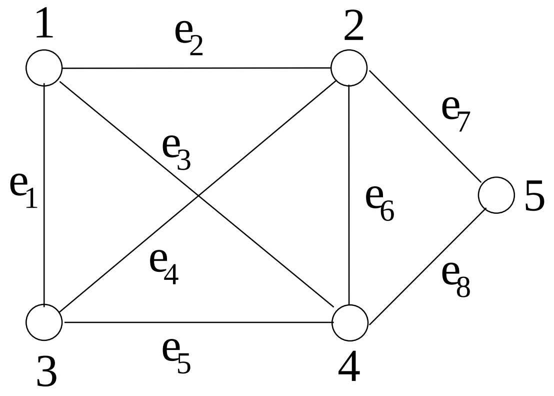
Terminology contd.

- A vertex v is called *isolated* in G if $\forall w \in V$ and $w \neq v$ there is no path between v and w,
- A graph is connected if there is path between any pair of vertices in G.
- For a subset $V_S \subset V$, the subraph induced by V_S is the grpah $S = (V_S, E_S)$ where $E_S = E \cap V_S \times V_S$.
- In other words, a subgraph of a given graph induced by a subset of vertices of the graph consist of all those edges which belong the graph and connect only the vertices of the subset.
- A connected component of a graph is the maximally connected subgraph of a graph.

Representation of graphs

- Adjacency matrix
- Edge incidence matrix
- Vertex-edge incidence matrix
- Adjacency lists

Example: adjacency matrix



$$A(i,j) = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & otherwise \end{cases}$$

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Example: edge-incidence matrix

$$IN(i, e_j) = \begin{cases} 1, & \text{if } i \text{ is an end vertex of} \\ & j\text{th edge } e_j \in E \\ 0, & otherwise \end{cases}$$

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Representing directed graphs

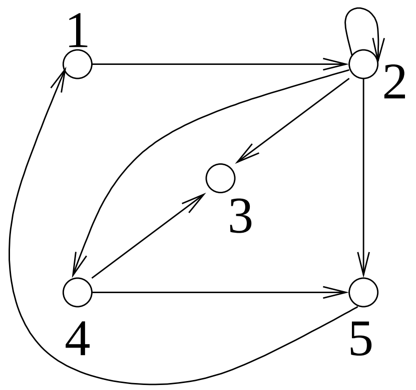
- Can be represented by both adjacency and edge incidence matrix.
- Only problem is to represent the orientation along with the edge.
- Orientation of edge is determined by some how specifying the initial vertex distinctly from terminal vertex.
- To achieve this is to define the adjacency matrix as follows

$$A(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise} \end{cases}$$

- As far as edge incidence matrix is concerned, define it as

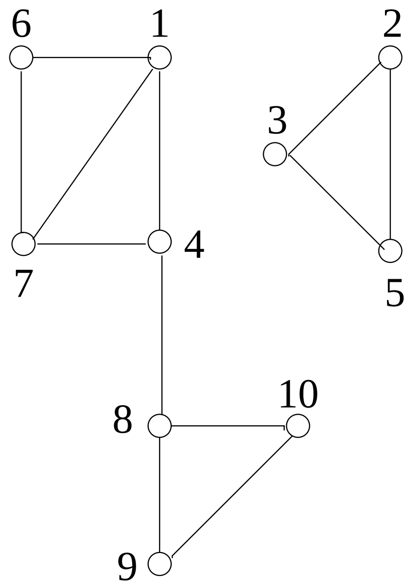
$$IN(i,e_j) = \begin{cases} 1 & \text{if } i \text{ is initial vertex of } e_j, \\ -1 & \text{if } i \text{ is terminal vertex of } e_j, \\ 0 & \text{otherwise} \end{cases}$$

Example: adjacency matrix



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Example: connected components



- Two connected components are subgraphs induced by
  - $\{1, 4, 6, 7, 8, 9, 10\}$, and
  - $\{2, 3, 5\}$

Graph objects and methods

- Objects: vertex, edge.
- Mutators:
  - `addEdge, addVertex, removeEdge, removeVertex`
- Accessors:
  - `isEdge, isConnected, isCyclic`.
  - `numberOfVertices, numberOfEdges, selectEdge, selectVertex`.
- Iterators:
  - `emanatingEdges, incidentEdges, vertices, edges`

Traversal or search

- Graph search is a systematic way to explore the graph, i.e., visit each vertex and traverse each edge at least once.
- Several ways to do that
  - Depth-first search
  - Breadth-first search
  - Breadth-depth search
  - k-depth search

Depth-first search

```
for (v ∈ V)
    color v white
while (there is a white vertex v)
    dfs(v)

algorithm dfs(v) {
    color v gray;
    for w ∈ L(v) {
        if (w is white) {
            set p(w) = v;
            dfs(w);
        }
        if (w is gray) and w ≠ p(v) then
            label (v, w) as back edge
        // if w is black it is not considered.
    }
    color v black;
}
```

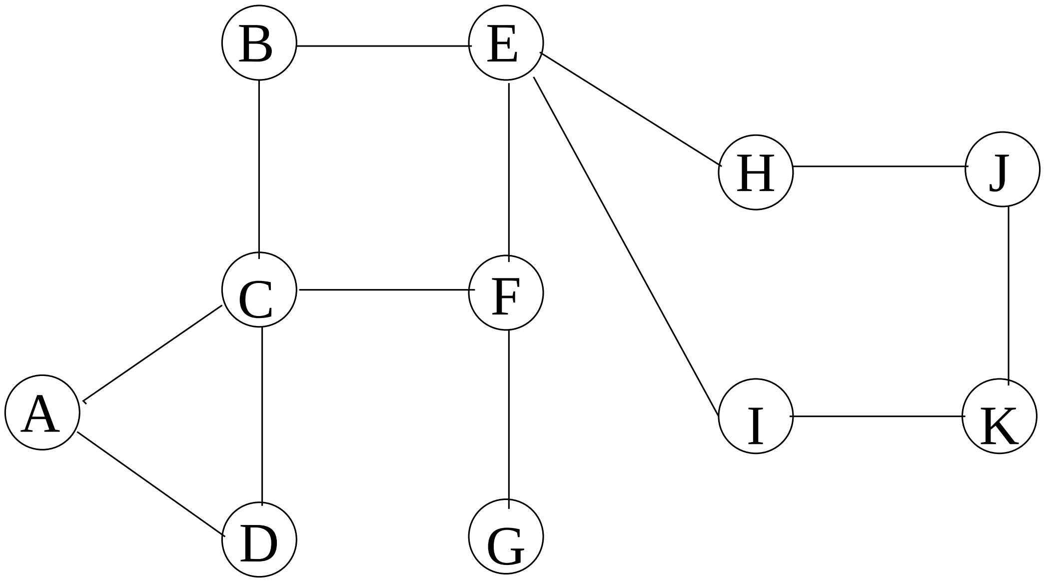
Depth-first search

- Rank vertices in DFS order visit – `start` & `finish` time
- DFS begins at v when v gets visited for the first time.
- Some edges are identified as `tree edges` and others as `back edges`.
- Consider only tree edges – each tree edge lead to a `white` vertex.

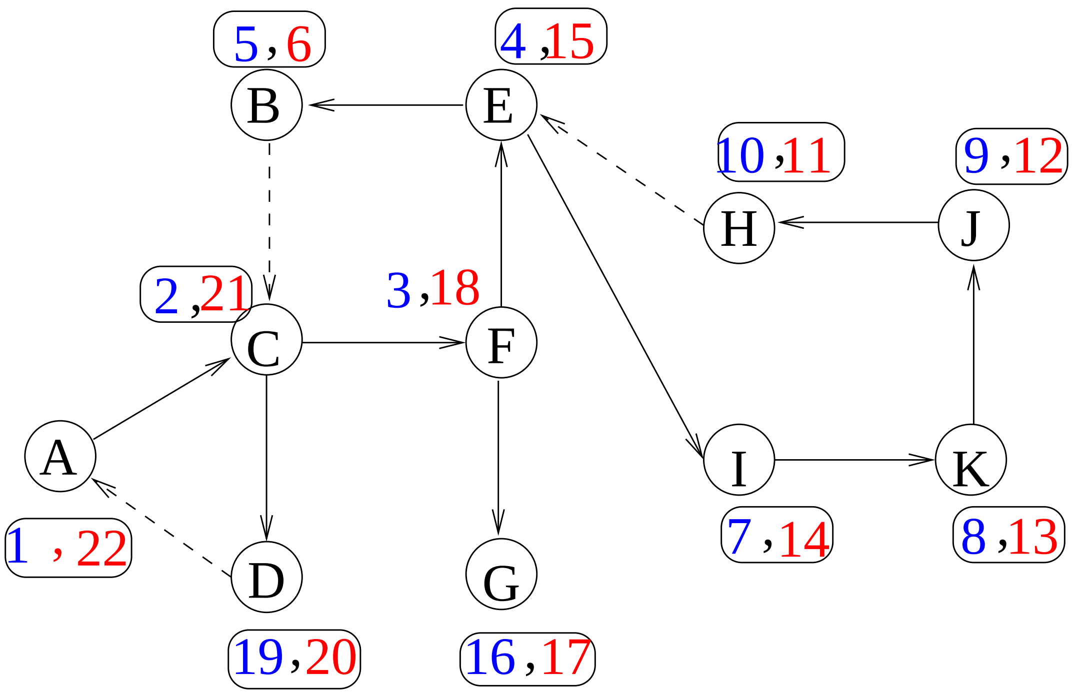
Depth-first search

- Each tree edge is examined twice – once *forward* and once *backward*.
- When traversed *forward* – white vertex is found,
- Increment DFS counter and assign value to the white vertex as start time.
- When tree edge about to be traversed *backward* – DFS at its terminal vertex is over.
- DFS counter also incremented and the value assigned as finish time of terminal vertex.
- Difference $DFS_{start}(v)$-$DFS_{finish}(v)$ should count number of tree edges in T_v two times.
- That is $2|E_{T_v}| = DFS_{start}(v)\text{-}DFS_{finish}(v) - 1$
- Thus, $\lceil (DFS_{start}(v)\text{-}DFS_{finish}(v))/2 \rceil$ is equal to $DESC(v)$.

Example: DFS



Example: DFS

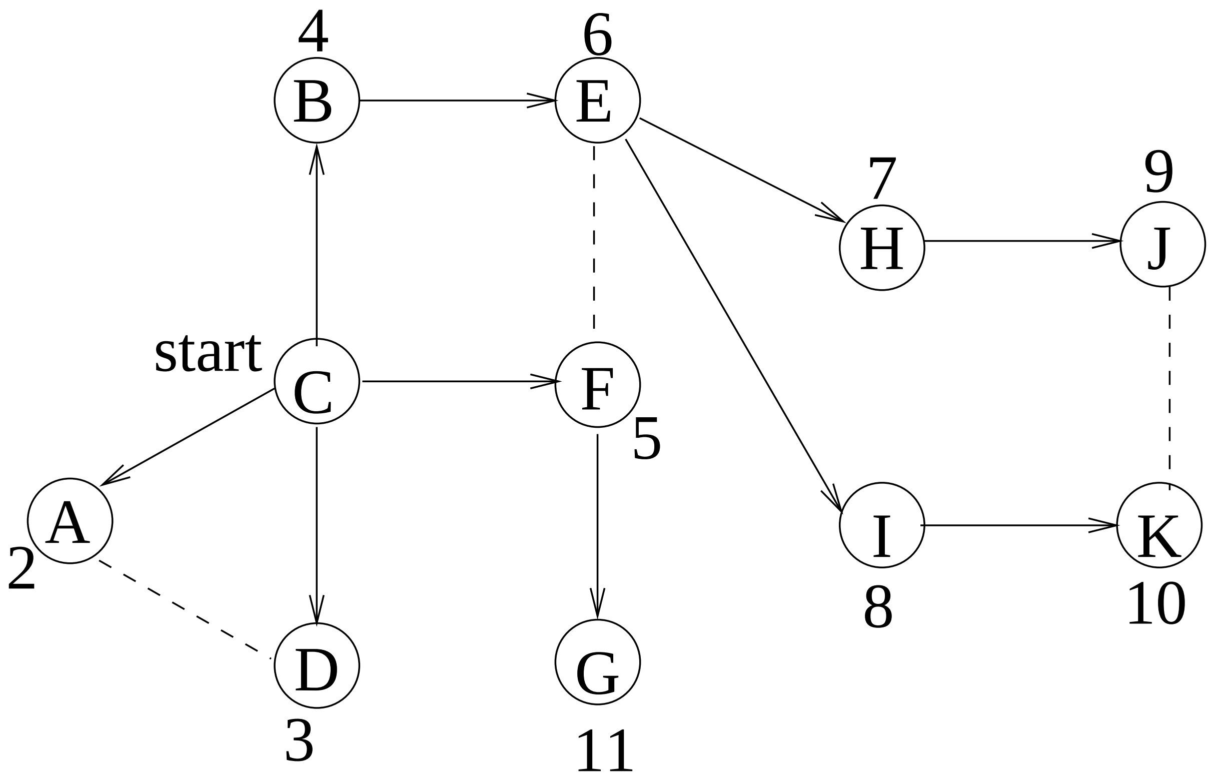


start time in red
finish time in blue

Breadth-first search

algorithm *Bfs*(G) {
    list L = empty
    tree T = empty
    choose a *starting* vertex x
    Search(x)
    while(L nonempty) {
        remove edge (v, w) from start of L
        if (w not yet *visited*) {
            add (v, w) to T
            Search(w)
        }
    }
}

Example: BFS



Example: BFS

BFS at	Vertices added to L	L
C	{A,D,B,F}	{A,D,B,F}
A	{}	{D,B,F}
D	{}	{B,F}
B	{E}	{F,E}
F	{G}	{E,G}
E	{H,I}	{G,H,I}
G	{}	{H,I}
H	{J}	{I,J}
I	{K}	{J,K}
J	{}	{K}
K	{}	{}

Depth-first search

algorithm *Dfs*(G) {
  list L = empty
  tree T = empty
  choose a starting vertex x
  Search(x)
  while(L nonempty) {
    remove edge (v, w) from end of L
    if (w not yet *visited*) {
      add (v, w) to T
      Search(w)
    }
  }
}

algorithm *Search*(vertex v){
  Visit(v);
  for each edge (v, w)
    add edge (v, w) to end of L
}

Connected components

- Can be found by modifying DFS only slightly.
- Initialize connected component number to 0.
- During recursive DFS only vertices of one connected component are visited.
- Therefore, increment connected component number only in the main driver program after a call to DFS terminates.

Connected component algorithm

algorithm *ConnectedComponent*(G) {
    $cn = 1$;
    for ($v \in V$) *unmark* v;
    while($v \in V$ is *unmarked*) {
        DFS(v);
        print current connected component;
        $cn++$;
    }
}

DFS(x) {
    mark x;
    for ($y \in L(x)$) {
        if (y is umarked) {
            add y to current connected component;
            DFS(y);
        }
    }
}

Depth-first search of directed graphs

- Depth-first search of directed graph can be carried out similarly.
- In case of directed graphs the edges are partitioned into 4 classes: **tree edges**, **back edges**, **forward edges**, **cross edges**.
- Classification of edges into 4 classes can be done during the DFS itself.
- But it is not a non-trivial problem.
- We analyze the relationships between the initial and terminal vertices of an edge to find out which class the edge may belong to.

Depth-first search of directed graphs

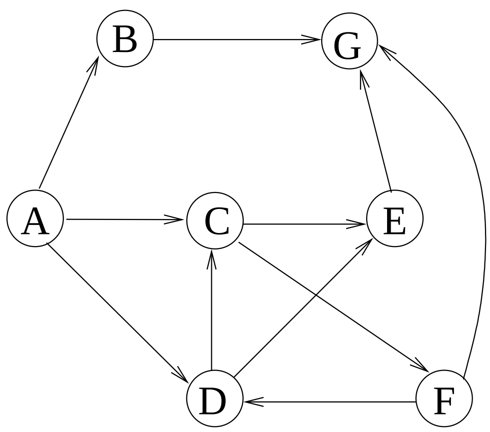
- Let $v \rightarrow w$ be a non-tree edge in a DFST.
- If $v \rightarrow w$ is either a **cross edge** or a **back edge** then w should be visited before v in DFS.
- If $v \rightarrow w$ is a **forward edge** then v should be visited before w in DFS – DFS at w must have finished.
- For a **back edge** w must be ancestor of v and for a **forward edge** v must be an ancestor of w – for a **back edge** DFS at w must be on-going.
- For a **cross edge** v and w are not related by ancestor descendant relations – DFS at w must have finished.

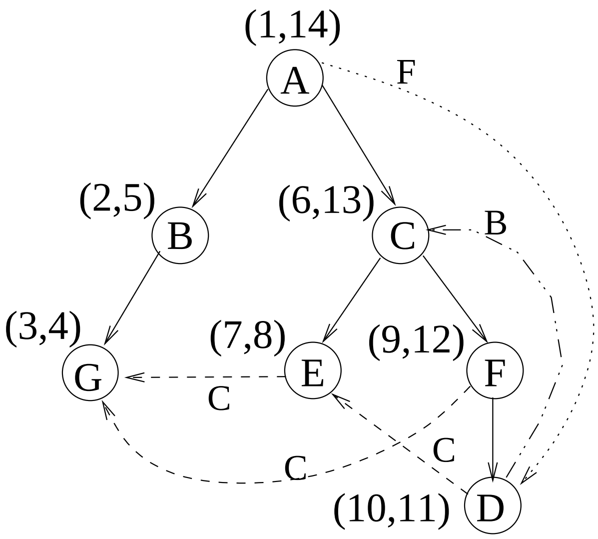
Classification of edges

```
algorithm dfs(v) {
    color v gray;
    // s(.) indicates start time
    s(v) = ++time;
    for w ∈ L(v) {
        if (w is white) {
            set p(w) = v;
            dfs(w);
        }
        if (w is gray)
            label (v, w) as back edge;
        else// f(.) indicates finish time
            if (f(w) < s(v))
                label (v, w) as cross edge;
            else if (p(w) ≠ v)
                    label (v, w) as forward edge;
    }
    color v black;
    f(v) = ++time;
}
```

Classification of edges

{ // Driver program for DFS
time=0;
for $(v \in V)$ color v *white*;
while $(v \in V$ such that v is *white*) *dfs(v)*;
}





Topological sort

- Topological ordering of vertices of a directed acyclic graph is important for application concerning PERT network.
- Each node represent a task and edges represent precedence relations.
- When we have a topological ordering the tasks can be processed in precedence preserving order.
- Topological ordering is not unique.
- We can accomplish it by slight change to DFS.

Topological sort

```
TS_DFS(x) {
    mark x;
    for (y ∈ L(x))
        if (y is umarked) TS_DFS(y);
    topLabel(x)=tLabel−−;
}

{ //Driver program
    tLabel = |V|;
    for (v ∈ V) unmark v;
    while(v ∈ V is unmarked)
        TS_DFS(v);
}
```

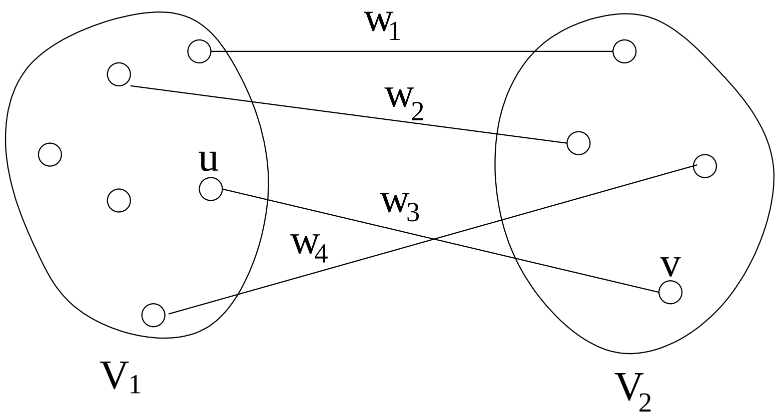
Minimum spanning tree

- Consider a weighted undirected graph $G = (V, E, w)$.
- An weighted graph consists of sets V and E and also associated with an additional function $w : E \to \mathbf{R}$
- A minimum spanning tree (MST) is a spanning tree T with the property that $\sum_{(u,v)\in T} w(u, v)$ is minimum.

Minimum spanning tree

- The property of MST can be restated more precisely as follows:

  Given any partition of vertices V into two non empty subsets V_1 and V_2, an MST contains only the minimum weighted edge (u, v) which connects a vertex $u \in S_1$ to a vertex $v \in S_2$. $w_3 = \min\{w_1, w_2, w_3, w_4\}$



Minimum spanning tree

- There are two distinct approaches to MST algorithms.
  - Start with disjoint collection of subtrees
  - Merge two subtrees by adding an edge of minimum cost between them.
  - Repeat the merging operation till the size of the collection reduces to 1 tree.
  - Start with a single vertex defining a tree
  - In each step add a new vertex to the tree such that edge of smallest cost joins it with a vertex already in the tree.

Minimum spanning tree

- The first approach of building a single tree from an initial collection of $|V|$ tree is proposed by Kruskal and usually referred to as Kruskal's algorithm
- The second approach of building MST from an initial one vertex tree is proposed by Prim and usually referred to as Prim's algorithm

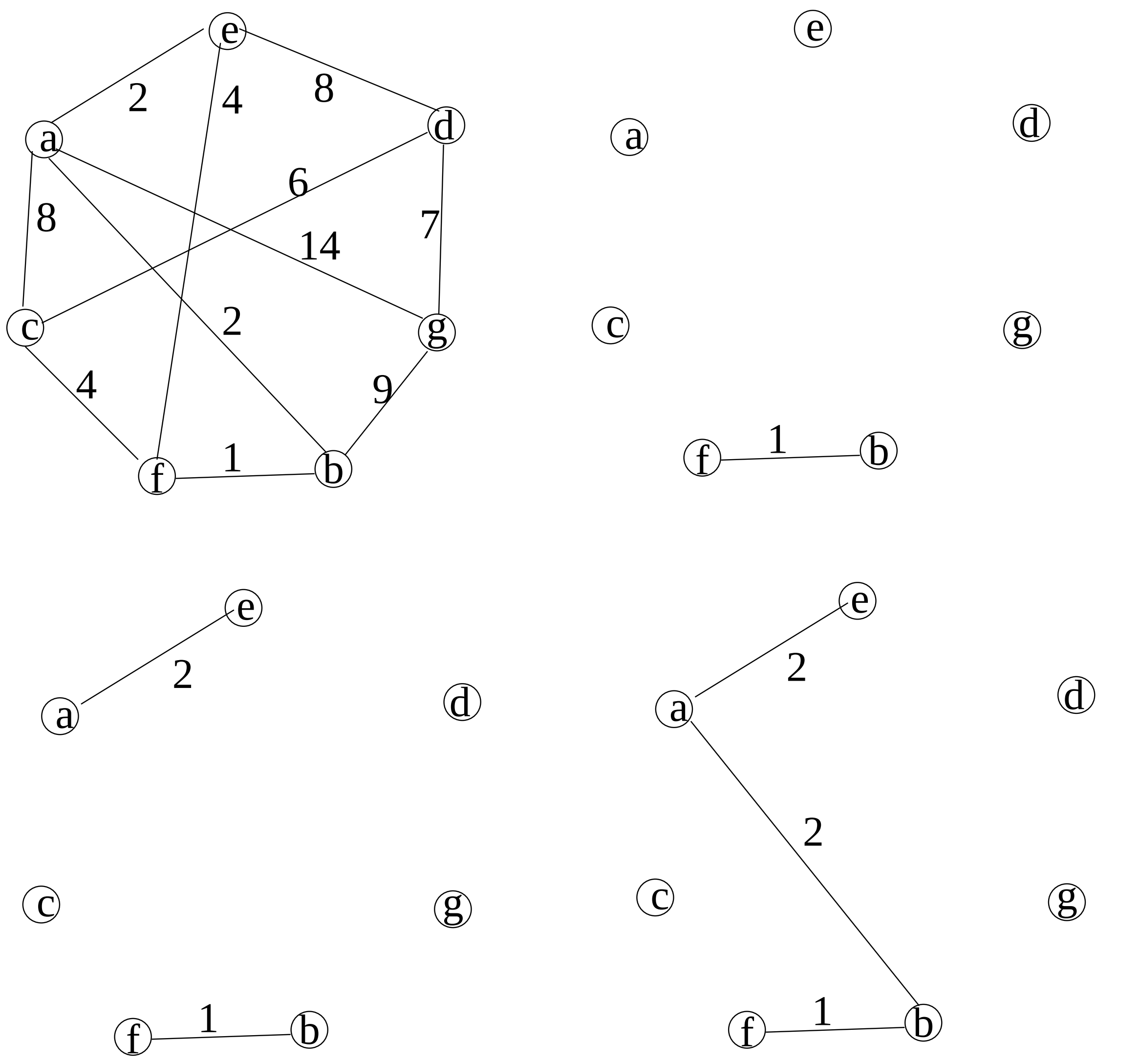
Kruskal's algorithm

T$=$ empty_set;
while (**card**$(T) < |V| - 1$) {
  $(u, v) =$ deleteMin(E);
  if ((u, v) does not form a cycle in T)
    add($(u, v), T$);
}

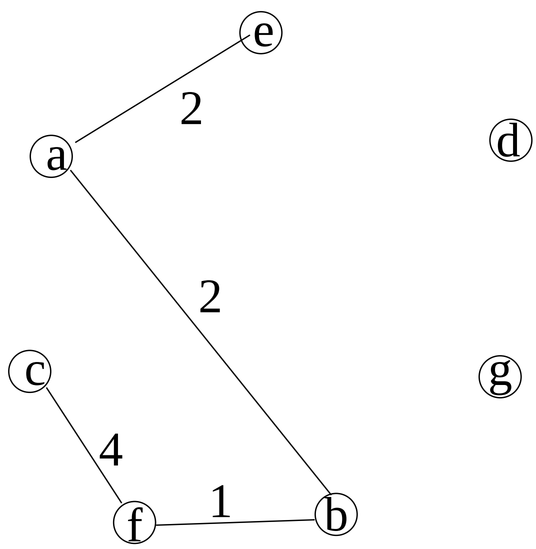
Kruskal's algorithm

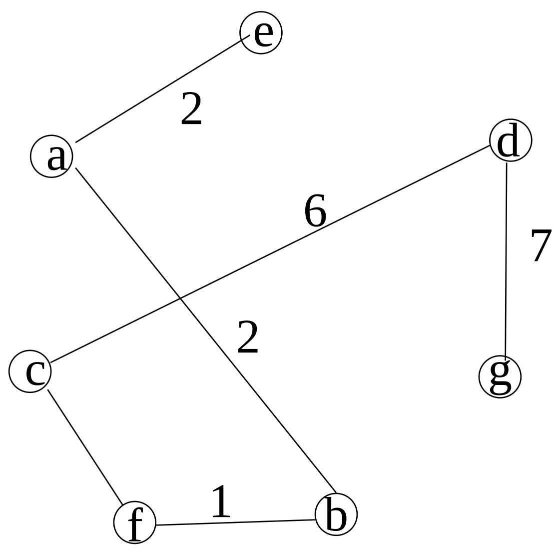
- Two main operations are: **cycle detection** and **delete minimum** wieghted edge
- DeleteMin can be executed in O($\log |E|$) time provided a heap of $|E|$ edges have been built already. As we know heap building requires time of O($|E|$).
- Cycle detection can be made easy provided we maintain the subtrees as disjoint sets. Because if the minimum weighted edge is found to join vertices of same tree them a cycle will result if that edge is retained in the set of spanning tree edges.

Kruskal's algorithm



Kruskal's algorithm





Algorithm: Kruskal

algorithm *Kruskal* {
    T= empty_set;
    buildHeap(H, E);
    while $(\mathbf{card}(T) < |V| - 1)$ {
        $(u, v) =$ deleteMin(H);
        $R_u =$ *find*(u);
        $R_v =$ *find*(v);
        if $(R_u \neq R_v)$ {
            union(R_u, R_v);
            add($(u, v), T$);
        }
    }
}

Prim's MST algorithm

- It begins with a single tree which initially just a vertex.
- Repeatedly adds a vertex not in the current tree such that the added vertex is connected by the least weight edge to a vertex in the current tree.
- So, logically we work with a partition of vertices of G into two sets.
- One set is T and the other set is its complement $V - T$.
- In each step one new vertex gets added to the set T and it complement decreases in size.
- When $|T| = |V|$ the algorithm terminates.

Prim's algorithm

algorithm *Prim* $\{$
    $T = \Phi$;
    $S = \{s : s \in V$ any node$\}$;
    $U = V - S$;
    for $(u \in U)$ $\{$
        $p[u] = s$; // node nearest to u
        $D[u] = w_{su}$; // weight of the edge (s, u)
    $\}$
    // Main loop here.
$\}$

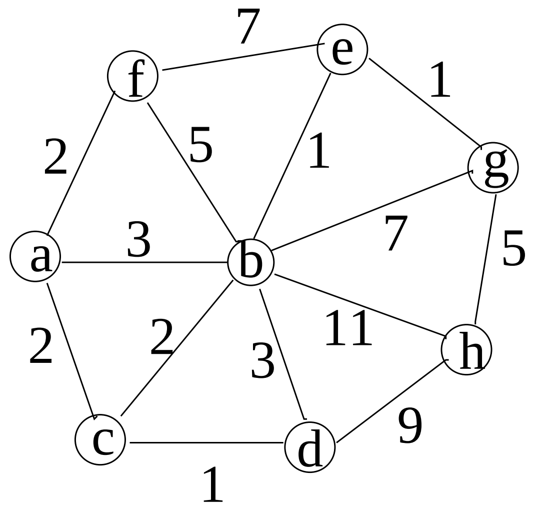
Prim's algorithm

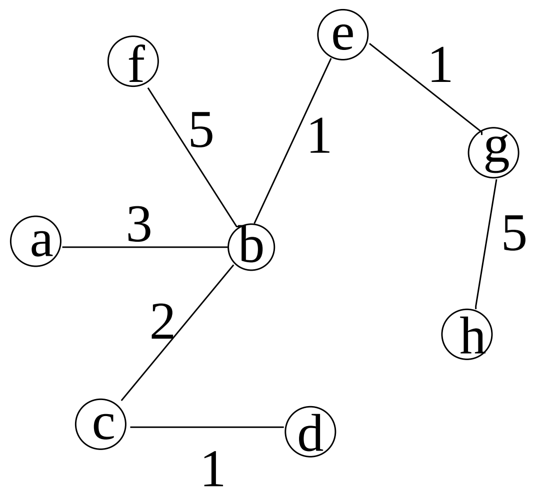
```
while (|S| < |V|) {
// minVertex(U) finds v ∈ U such that
// D[v] = min{D[u]|u ∈ U}
    v = minVertex(U);
    add(v, S);
    delete(v, U);
    add((v, p[v]), T);
    for (each node u ∈ U) {
        if (D[u] > w_uv) {
            D[u] = w_uv; // update weights
            p[u] = v; // update predecessor
        }
    }
}
```

Shortest paths

- Shortest path has many applications.
- Two important shortest problems:
  - single source shortest path,
  - all pairs shortest paths.
- Single source shortest paths consists of collection of shortest paths from a source (a single vertex) to all other vertices.
- Subpath of a shortest path will also be a shortest path.
- So the collection of shortest paths form a tree.
- Tree is may not be unique.

Shortest paths





Shortest paths

- Dijkstra discovered this algorithm.
- Each vertex receives two different labels: temporary and permanent.
- When vetex has a temporary label it may be relabeled.
- After a temporary label declared permanent, no further relabeling.
- Start with source being assigned permanent label of 0.
- Choose the vertex with least temporary label and make it permanent.
- Update temporary labels of all vertex v such that $(x, v) \in E$ recent most permanentally labeled vertex x as follows;
  - if temporary label of v is greater than permanent label of x plus $w(x, v)$ then resent temporary label of v to new value.
- Continue until all vertices are permanently labeled.

Shortest paths

vert	T label	P label
a	3	
b	0	0
c	2	
d	3	
e	1	
f	5	
g	7	
h	11	

vert	T label	P label
a	3	
b	0	0
c	2	
d	3	
e	1	1
f	5	
g	$\boxed{2}$	
h	11	

vert	T label	P label
a	3	
b	0	0
c	2	2
d	3	
e	1	1
f	5	
g	2	
h	11	

vert	T label	P label
a	3	
b	0	0
c	2	2
d	3	
e	1	1
f	5	
g	2	2
h	$\boxed{7}$	

Shortest paths

vert	T label	P label
a	3	
b	0	0
c	2	2
d	3	3
e	1	1
f	5	
g	2	2
h	7	

vert	T label	P label
a	3	3
b	0	0
c	2	2
d	3	3
e	1	1
f	5	
g	2	2
h	7	

vert	T label	P label
a	3	3
b	0	0
c	2	2
d	3	3
e	1	1
f	5	5
g	2	2
h	7	

vert	T label	P label
a	3	3
b	0	0
c	2	2
d	3	3
e	1	1
f	5	5
g	2	2
h	7	7

Dijkstra's algorithm

$pLabel[s] = 0$;
for each $v \neq s \in V$ do {
    if $(v \in L[s])$
        $tLabel[v] = w(s, v)$;
    else
        $tLabel[v] = +\infty$;
    $pLabel[v] = un$;
}
for $|V| - 1$ iterations do {
    find u such that $pLabel[v] = un$ and
        $tLabel[u] = \min\{tLabel[v]\}$
    $pLabel[u] = tLabel[u]$;
    for each $v \in L[u]$ do
        if $(pLabel[u] + w(u, v] < tLabel[v])$
            $tLabel[v] = pLabel[u] + w(u, v)$;
}
return $pLabel[v]$ for all $v \in V$;

Shortest paths

- Analyzing the algorithm in present form: the running time is O($|V|^2$).
- Note that at ith iteration $pLabel$s of exactly i vertices are undefined.
- Thus in iteration i
  - One vertex gets permanent label – O(1) time, and
  - Finding the vertex with minimum temporary label among i vertices – $\sum_{1}^{i=|V|-1} i$
  - In worst case, temporary labels of $i-1$ adjacent vertices are updated – $\sum_{1}^{i=|V|-2} i$.
- Therefore, time complexity is O($|V|^2$).

Shortest paths

- We can speed up the algorithm considerably by arranging the temporary labels in a heap.
- The minimum label will be on the top of heap.
- Update of temporary labels will require at most O($|L(v)|$) heapify operations.
- Each update can be considered as a *decreaseKey* operation. which can be accomplished in $\log |V|$ time provided we implement the heap using an array.
- Thus E updates would need O($E \log V$) time resulting an overall running time of O($E \log V$).

All pairs shortest paths

- Running single source shortest path algorithm $|V|$ times results in running time of O($VE\log V$).
- For dense graphs $|E| = O(|V|^2)$. So the running time is O($V^3 \log V$).
- But we should be able to do better than that.
- Floyd-Warshall's algorithm achieves a time bound of $\Theta(V^3)$.
- It implies that for dense graphs it performs better than running single source shortest path $|V|$ times.

Floyd-Warshall's algorithm

- A vertex v_i in a simple path $P = \{v_1, v_2, \ldots, v_\ell\}$ is an intermediate vertex if $v_i \in \{v_2, v_3, \ldots, v_{l-1}\}$.
- Floyd-Warshall's algorithm is based on the following idea:

  Let, P be the path of minimum weight among all the paths from v_i to v_j which can only include $\{v_1, v_2, \ldots, v_{k-1}\}$ as intermediate vertices. Suppose, v_k is also included as an intermediate vertex then it may be possible get a path shorter than P.
- When v_k occurs as an intermediate vertex on P then we can break it into two parts : $P_1 : v_j \Rightarrow v_k$ and $P_2 : v_k \Rightarrow v_i$.
- P_1 and P_2 are shortest paths with intermediate vertices drawn from $\{v_1, v_2, \ldots, v_{k-1}\}$.

Floyd-Warshall's algorithm

- In view of the above observation a recursive solution to APSP is defined as follows

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

- $d_{ij}^{(0)}$ represents the length of a path from v_i to v_j with no intermediate vertex, and therefore, it just consists of the direct edge from v_i too v_j implying $d_{ij}^{(0)} = w_{ij}$.
- By computing $D^{(n)} = \left(d_{ij}^{(n)}\right)$

Construction of shortest path

- Actual path construction can be done on-line during execution of Floyd-Warshall's algorithm.
- We can define a path matrix P as follows:

$$P^{(0)} = \left(p_{ij}^{(0)}\right)$$

$$p_{ij}^{(0)} = \begin{cases} nil & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

Updating path matrix

- Updating path matrix so that it can find the predecessor of any intermediate vertex on the shortest path between any pair of vertices is quite simple.
- It is provided by the following recursive definition.

$$p_{ij}^{(k)} = \begin{cases} p_{ij}^{(k-1)} & \text{if } d_{ij}^{(k)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ p_{kj}^{(k-1)} & \text{if } d_{ij}^{(k)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Example for Floyd-Warshall's algo