



Graph



Graph - Definitions

- ❖ A graph G is represented by a tuple (V, E) where V is a set of vertices $V=(v_1, v_2, v_3, v_4, \dots)$ and E is a set of edges $E=(e_1, e_2, e_3, e_4, \dots)$
- ❖ An element of E , say e_i is a pair of vertices (v_m, v_n) .
So $E \subseteq V \times V$
- ❖ If $e_i = (v_m, v_n)$ is an ordered pair, then the graph is a directed one; v_m is the start vertex and v_n is the end vertex. We call such graphs as Digraphs.
- ❖ If $e_i = (v_m, v_n)$ is an edge of G , then v_m and v_n are said to be adjacent.

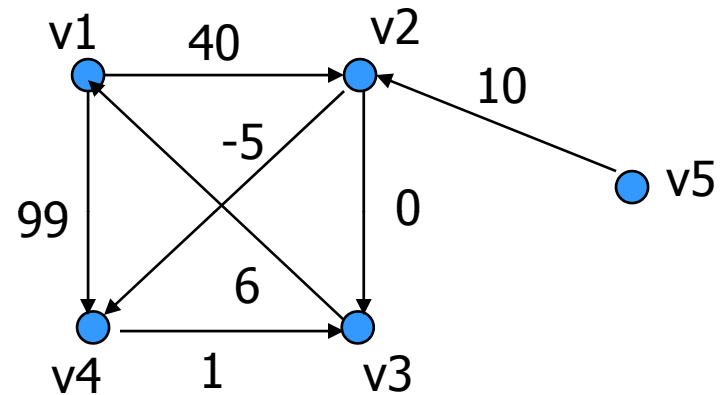
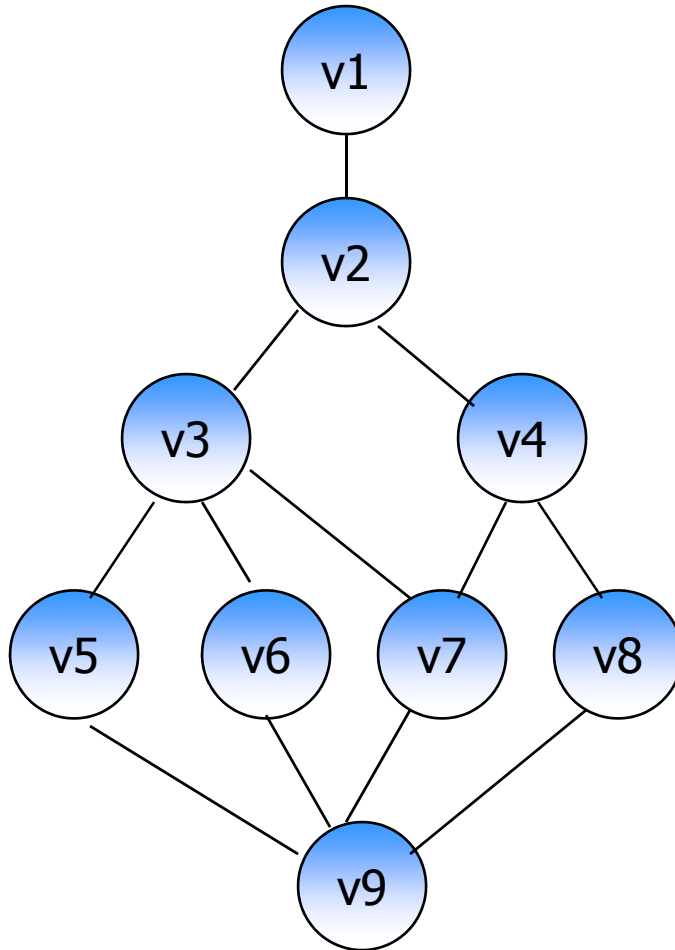


Graph – Definitions ...

- ❖ The number of edges incident on a vertex is said to be degree. In a digraph, in-degree and out-degree are differentiated.
- ❖ A graph $G' = (V', E')$ is called a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- ❖ A path from vertex v_1 to v_n is a sequence of vertices $v_1 v_2 v_3 \dots v_k$, such that (v_i, v_{i+1}) is an edge for $i = 1$ to $k-1$.
- ❖ Paths may have cycles and edges may have associated weights.



Examples





Graph Representation – Adjacency Matrix

- Let $G = (V, E)$ be a graph of n vertices. Its adjacency matrix is an $n \times n$ matrix M consisting of 0s and 1s.
- $M[i][j] = 1$ iff (v_i, v_j) is an edge of G
 $= 0$ otherwise
- For undirected graphs, adjacency matrix is necessarily symmetric.
- For a weighted graph,
 $M[i][j] = \text{weight}((v_i, v_j))$, if $(v_i, v_j) \in E$
 $= \text{infinity}$, otherwise



Adjacency Matrix

```
#define maxnovertices 100
```

```
typedef struct {  
    int    novertices;  
  
    float  M[maxnovertices][maxnovertices];  
} graph;
```



Graph Representation – Adjacency List

Here a list is maintained for each vertex.

The list for any vertex contains the vertices adjacent to it.

For weighted graphs, additionally, the weight information is stored.



Adjacency List

```
#define      maxnovertices      100

typedef struct node      {
            int      to_vertex;
            float      weight;
            struct node      * neighbour;
            }      nodetype;
typedef      struct {
            int      novertices;
            nodetype      * List[maxnovertices];
            }      graph;
```




Adjacency List ...

Total number of nodes required for a graph (undirected) of n vertices and e edges is $n+2e$.

For digraph, the number is $n+e$.

For sparse graph, this representation is efficient.



Graph Traversal – Depth-First Search

```
rec-dfs (v)
{
    visit(v);
    for (each vertex u adjacent to v)
    if (u is not yet visited)
        rec-dfs(u);
}
```



Iterative Depth-First Search

```
iter_dfs ( graph v){  
    int                i,u1,u2;  
    stack-of-vertices  stak;  
    unsigned char      visited[maxnovertices];  
  
    for (i=0; i< maxnovertices; i++) visited[i] = false;  
    s_create(stak);  
    push(v, stak);  
    do{  
        u1 = pop(stak);  
        if (!visited[u1]) {visit(u1); visited[u1] = true;}  
        for (each u2 adjacent to u1){  
            if (!(visited[u2])) push(u2, stak);}  
    }while (! empty(stak));  
}
```



Graph Traversal – Breadth-First Search

```
BFS(graph v){  
    int                i,u1,u2 ;  
    queue-of-vertices  q;  
    unsigned char      visited[maxnovertices];  
  
    for (i=0; i< maxnovertices; i++) visited[i] = false;  
    init-q(q); enqueue(v, q);  
    do{  
        u1 = dequeue(q);  
        if (!(visited[u1])) {visit(u1); visited[u1] = true;}  
        for (each u2 adjacent to u1){  
            if (!(visited[u2])) enqueue(u2, q);}  
    }while (!empty_q(q));  
}
```

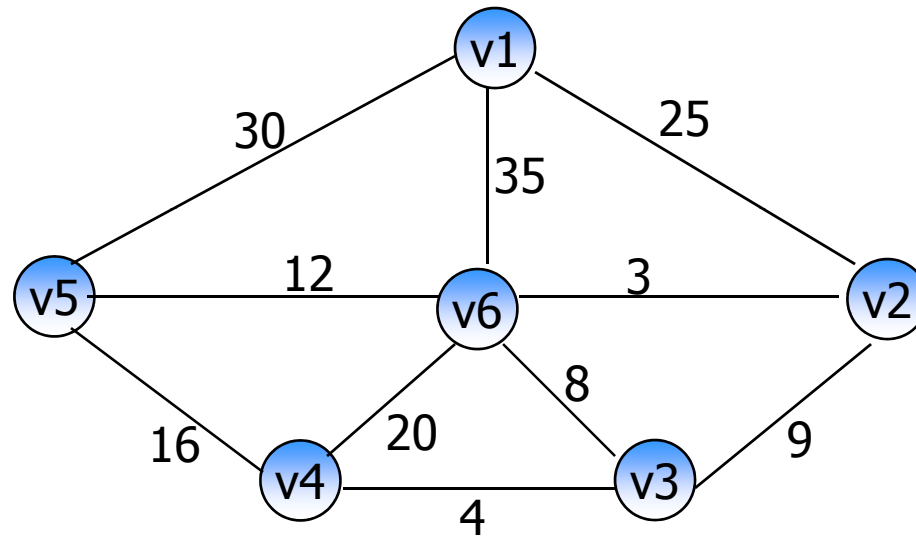


Spanning Tree

- ❑ A subgraph G' of G which contains all the vertices of G and is a tree is called a spanning tree of G . There may be more than one spanning trees of a graph.
- ❑ In case of a weighted graph one can assign a cost to a spanning tree, defined as the sum of the weights of its edges.
- ❑ An important problem is to find out the minimum cost spanning tree (MCST) of a graph which may not be unique.



Find the Minimum Cost Spanning Tree





Kruskal's Algorithm

```
MCST-K(graph G){           //G=(V,E)
    tree  T;
        init_t(T);
        sort E in non-decreasing order according to the weight of
        the edges;
        while(T contains less than (n-1) edges and E is not empty) {
            e = edge(u,v) ∈ E having least weight;
            delete e from E;
            if (inclusion of e in T does not form a cycle) add e to T;
        }
        if (T contains less than (n-1) edges)
            error ("no spanning tree exists")
        else      output T as the MCST;
    }
```

Complexity $O(|E| \log |E|)$



Prim's Algorithm

```
TV = { v1};  
for (T=Φ; T contains fewer than n-1 edge; add (u,v) to T)  
{  
    Let (u,v) be a least-cost edge such that u ∈ TV and !(v ∈ TV);  
    if (there is no such edge) break;  
    add v to TV;  
}  
if (T contains fewer than n-1 edges)  
    print (“no spanning tree exists”);
```




Dijkstra's shortest path algorithm (Greedy)

SP-D(G)

//Find the shortest paths from v_1 to all the vertices

{

$l_1^* = 0;$

for ($i = 2; i \leq n; i++$) $l_i = C_{1i};$

do{

$l_k = \min \{l_i\};$ // among all temporary labels

$l_k^* = l_k;$

Let v_k be the node which just received a permanent label;

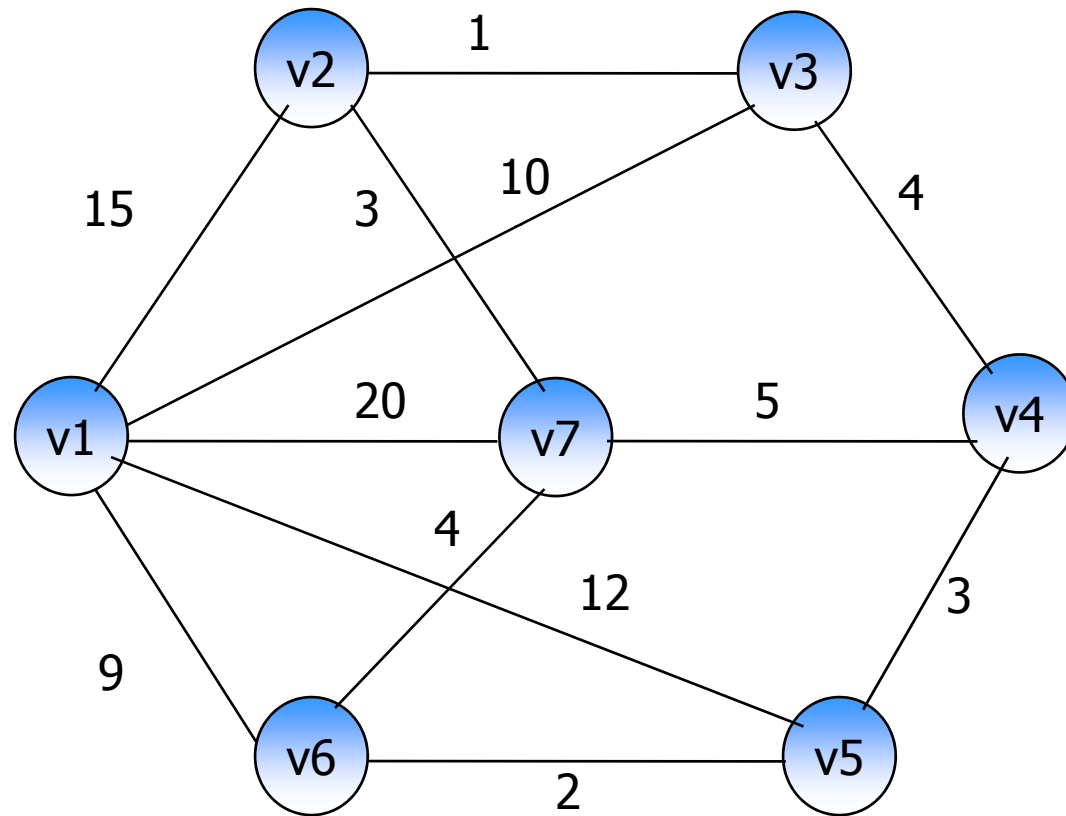
for all adjacent vertices of v_k having temporary labels do

$l_i = \min(l_i, l_k^* + C_{ki});$

while there are vertices without permanent labels;

}

Time Complexity = $O(n^2)$





All pair Shortest Paths

```
APSP (graph G){  
  float A[n][n];  
  for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
      A[i][j] := M[i][j];  
  for (k = 0; k < n; k++)  
    for (i = 0; i < n; i++)  
      for (j = 0; j < n; j++)  
        A[i][j] = min(A[i][j], A[i][k] + A[k][j]);  
}
```

Time Complexity = $O(n^3)$



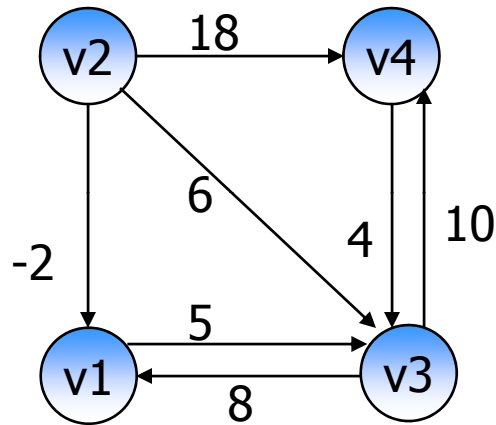
Transitive Closure Matrix

```
APSP (graph G){  
  boolean T[n][n];  
  for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
      if (M[i,j] <>  $\infty$ ) T[i][j] = true else T[i][j] = false;  
  for (k = 0; k < n; k++)  
    for (i = 0; i < n; i++)  
      for (j = 0; j < n; j++)  
        T[i][j] = T[i][j] OR ( T[i][k] AND T[k][j])  
}
```

Transitive Closure Matrix – A matrix of boolean values where a true at (i,j) represents the existence of a path between nodes i and j



Digraph with cycles and negative weight



	v1	v2	v3	v4
V1	0	∞	5	∞
V2	-2	0	6	18
V3	∞	8	0	10
V4	∞	∞	4	0