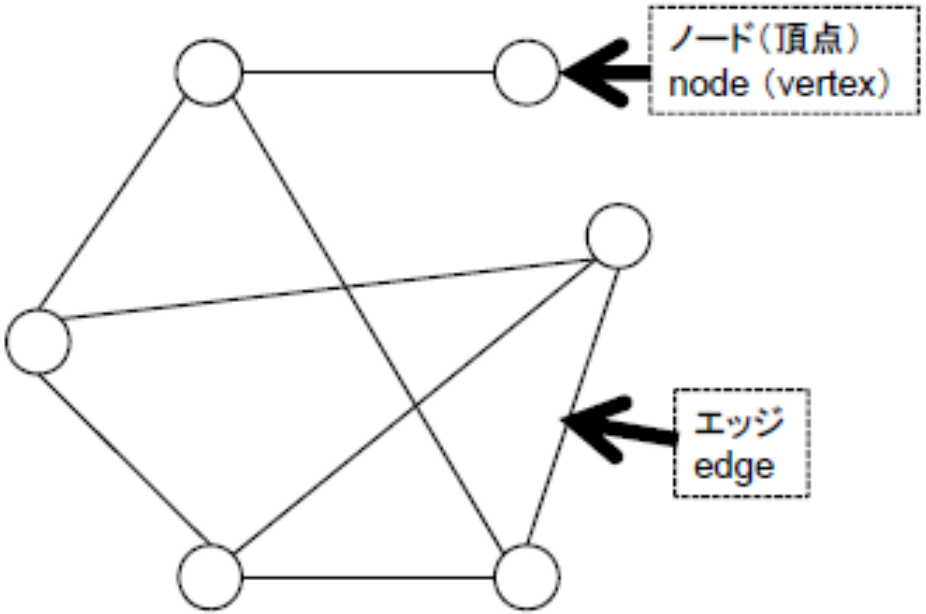
GRAPH

グラフ(graph)

- ノード(頂点)(node/vertex)の集合とエッジ(edge)の集合から構成される
  - 無向グラフ(undirected graph):エッジが方向性をもたないグラフ
  - 有向グラフ(directed graph):エッジが方向性をもつグラフ



Graph

Graph is a collection of vertices (or nodes) and the connections between them.

- It represents one limitation of trees. Trees can only represent relations of hierarchical type, such as relations between parent and child.

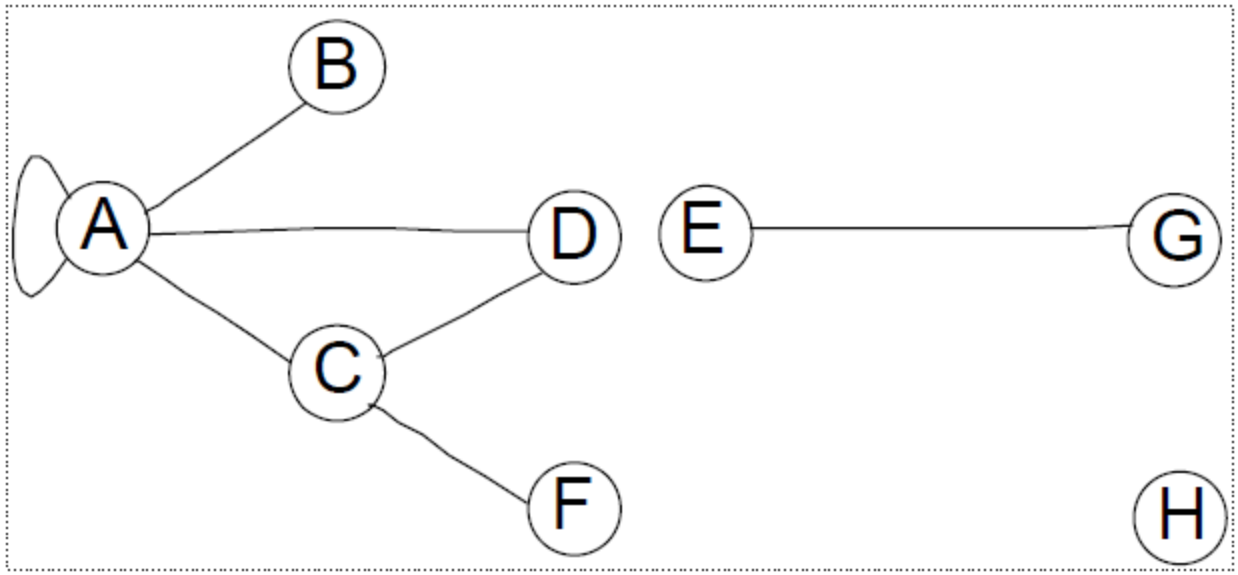
Review: Graph Theory

- **Graph** – discrete structures consisting of vertices and edges that connect these vertices.
- A graph G = (V, E) consists of V, a nonempty set of **vertices** (or nodes ) and E, a set of **edges**. Each edge has either one or two vertices associated with it, called its **endpoints**.

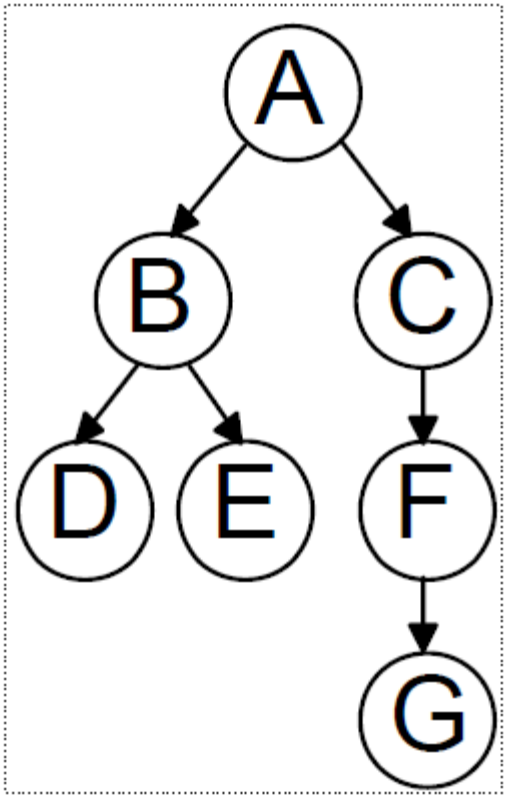
Applications:

- Networks (LAN, MAN, Social Networks, etc.)
- Job assignments
- Representing computational models

Example:



Undirected Graph



Directed Graph

Note:

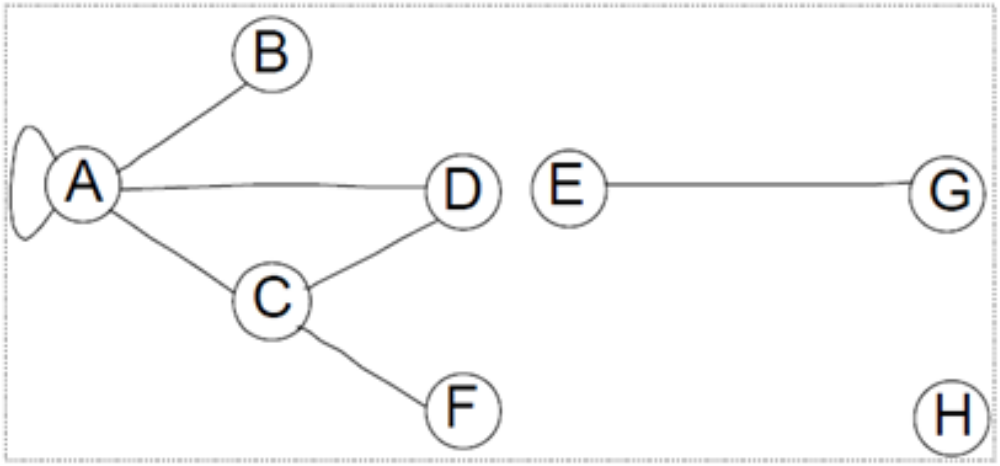
1. A graph need not be a tree but tree must be a graph.
2. A node need not have any arcs associated with it.

Directed Graph

- If the pairs of nodes that made up the arcs are ordered pairs the graph is said to be directed graph or digraph.

Basic Terminology

1. Arcs (or edges)
   - Connections between nodes or vertices.

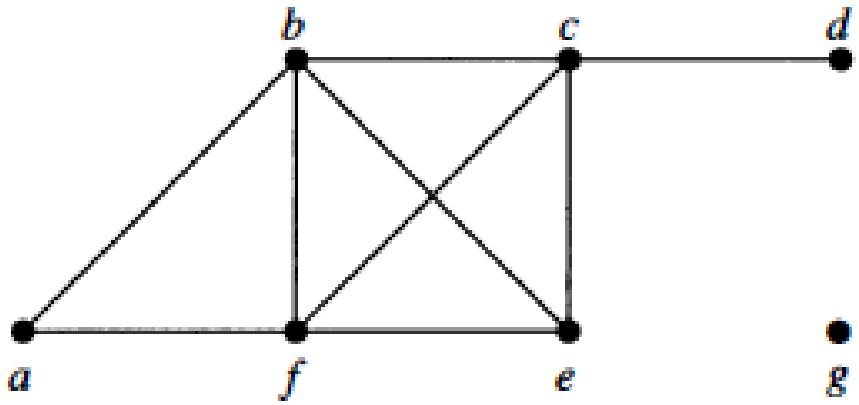


Set of nodes: {A,B,C,D,E,F,G,H}

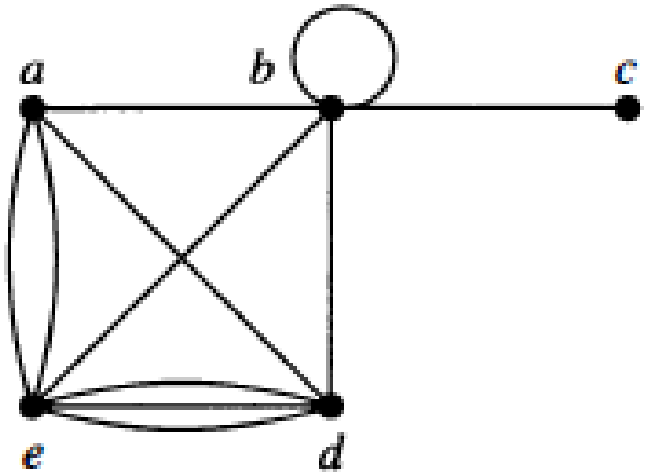
Set of arcs:
{(A,B),(A,D),(A,C),(C,D),(C,F),(E,G),(A,A)}

Basic Terminology

- The **degree** of a vertex in an undirected graph is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex.



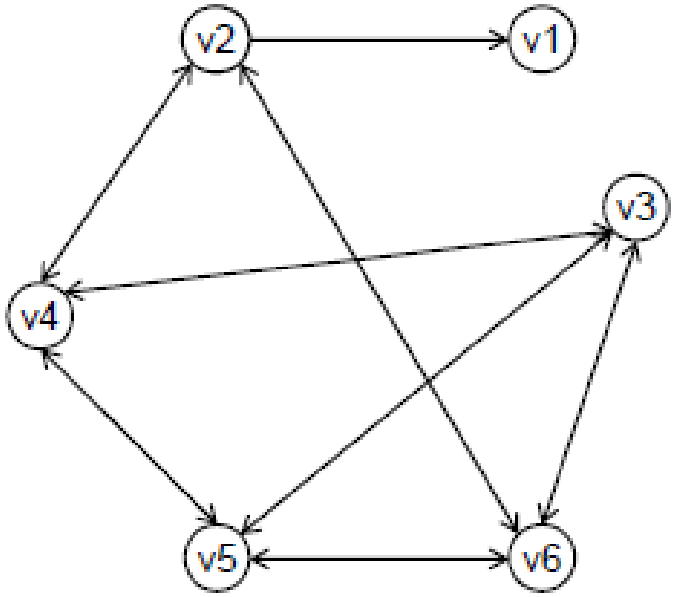
deg(a) = 2 , deg(b) = deg(c) = deg(f) = 4 deg(d) = 1 , deg(e) = 3 , and deg(g) = 0.



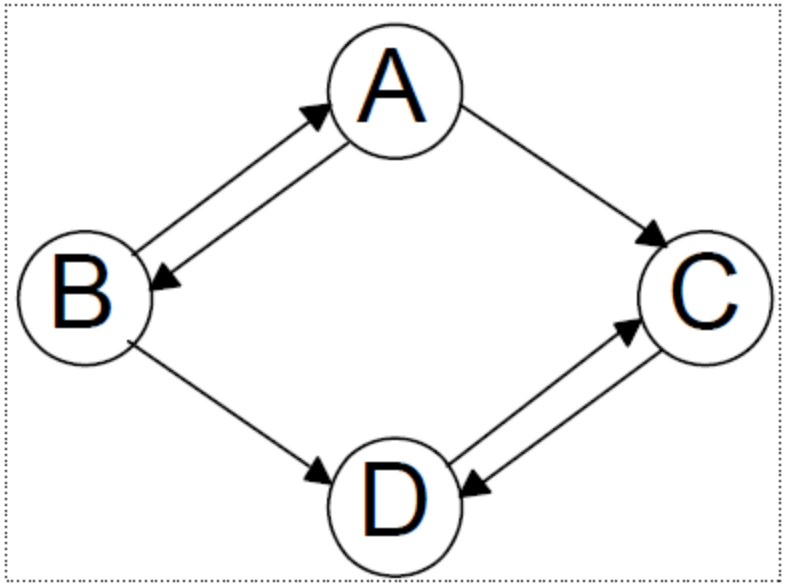
deg(a) = 4, deg(b) = deg(e) = 6, deg(c) = 1 , and deg(d) = 5

次数（degree）

- グラフ中のあるノードが持つエッジの数
  - 例：v2の次数は3
- グラフが有向グラフの場合には，エッジの向きを区別する
  - 入次数（indegree）：あるノードに入ってくるエッジの数
  - 出次数（outdegree）：あるノードから出ていくエッジの数



Example



Degree of A = 3

Indegree of A = 1

Outdegree of A = 2

Indegree

- The indegree of a node n is the number of arcs that have *n* as the head.

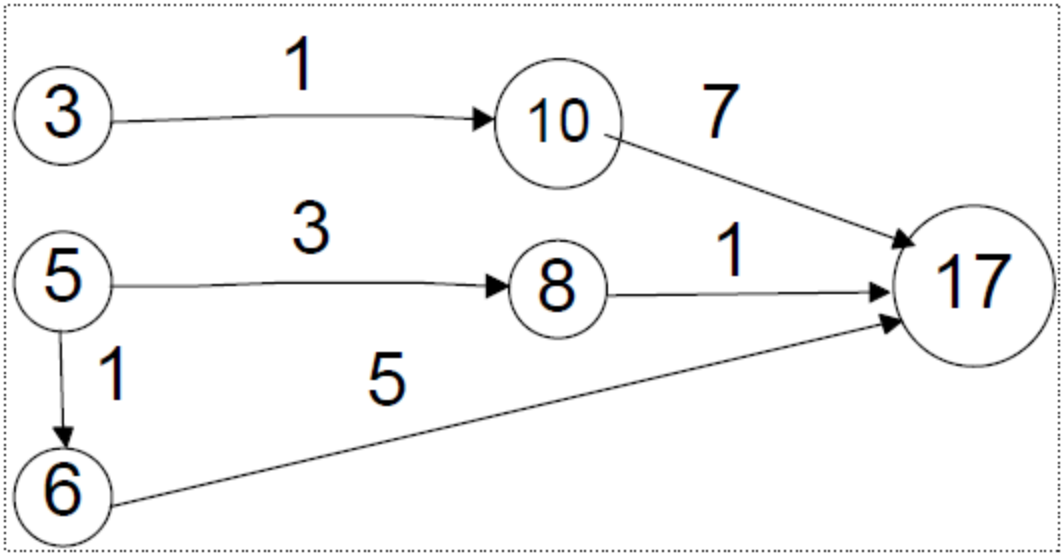
Outdegree

- The outdegree of a node n is the number of arcs that have *n* as the tail.

Basic Terminology

Relation

- A relation R on a set A is set of ordered pairs of elements of A.



Set A: {3,5,6,8,10,17}

Set of R: {<3,10>,<5,6>,<5,8>,<6,17>,<8,17>,<10,17>}

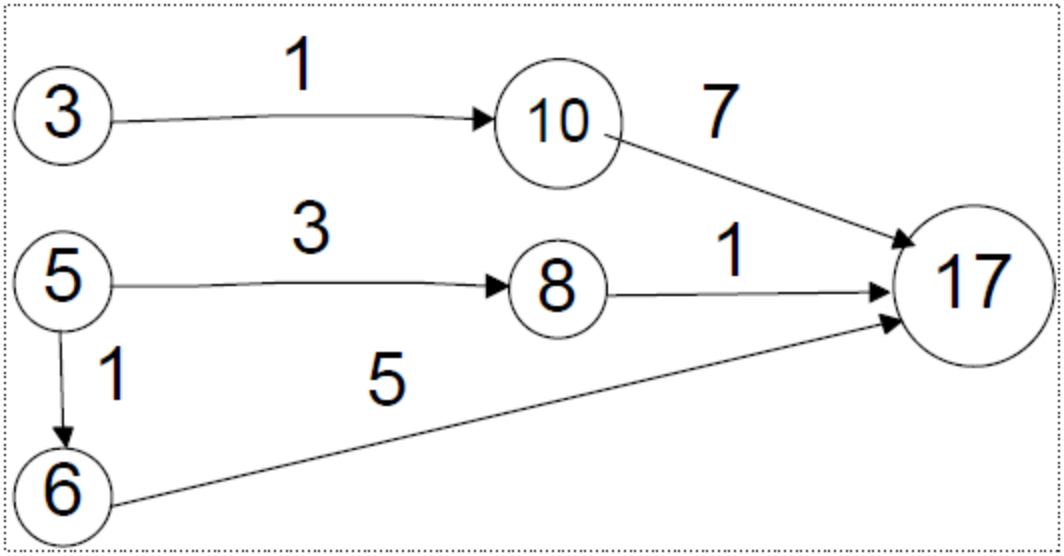
Basic Terminology

Weighted Graph or Network

- A graph in which a number is associated with each arc.

Weight

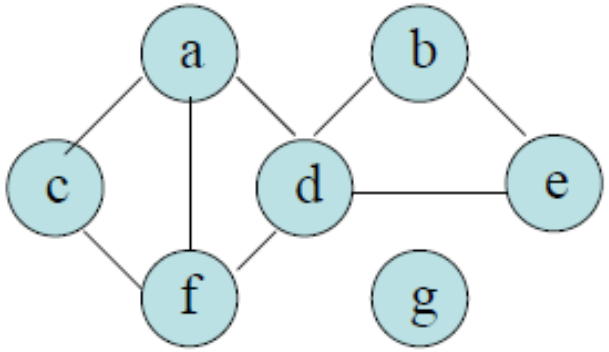
- The number associated with an arc.



Weight= (Head node value) mod (Tail node value)

Graph Representation

Example:



Equivalent Graph Representation:

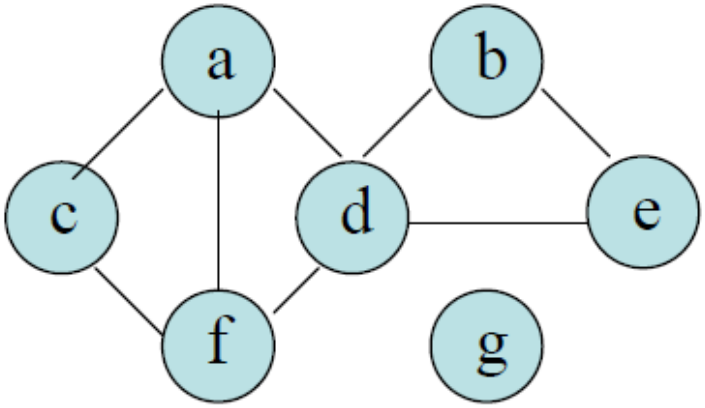
Adjacency – Two nodes are called adjacent if the edge formed by this two nodes is in the set of edges.

Adjacency List using Array

a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

Drill

- Find the adjacency matrix of the following graph:



	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

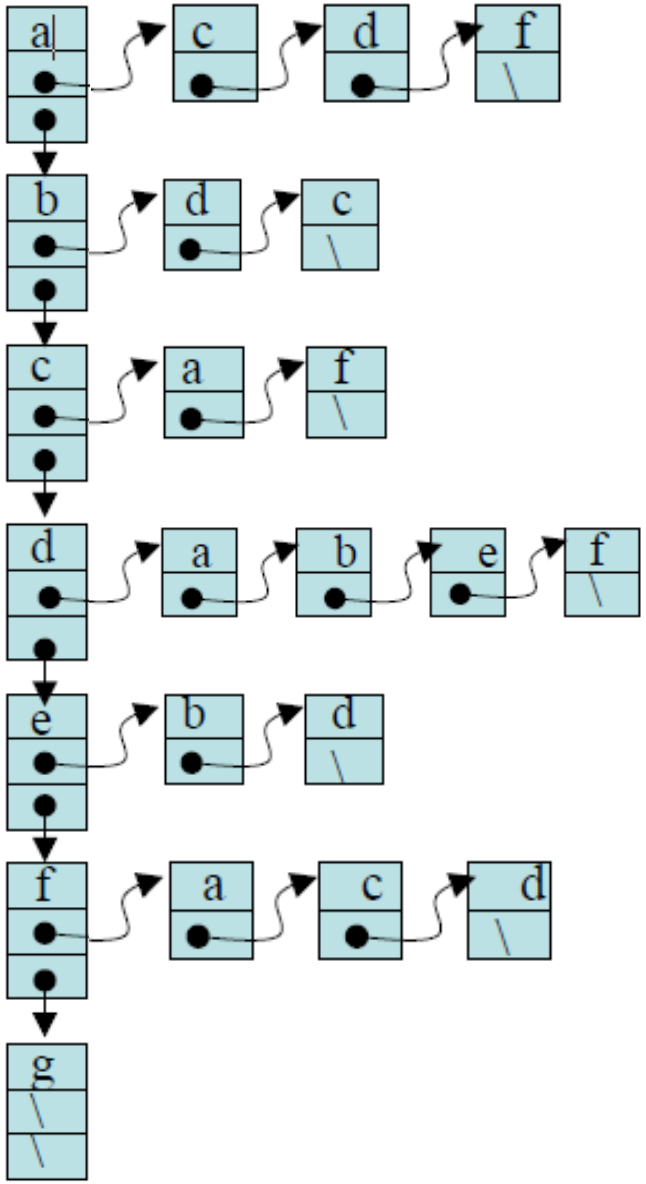
Adjacency Matrix – it represents every possible ordered pair of nodes.

Graph Representation

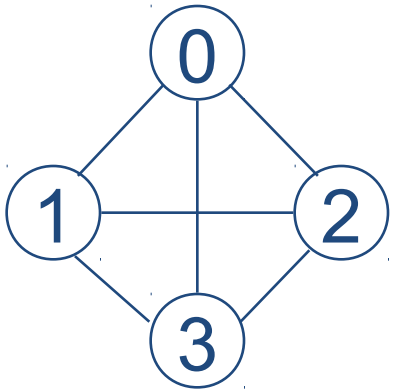
- Adjacency list

using Linked list

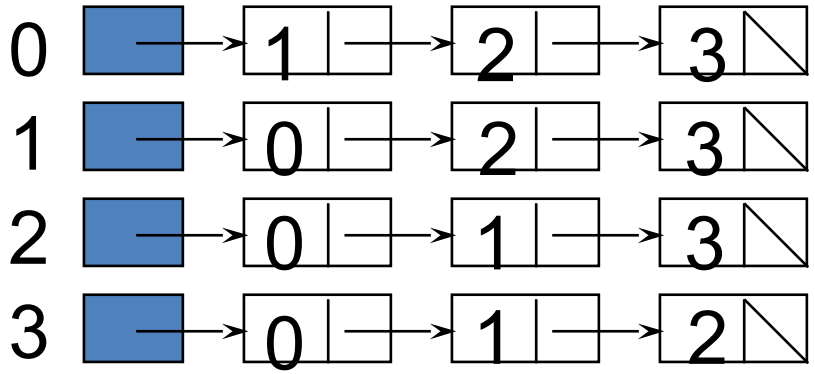
Adjacency List – it specifies all nodes adjacent to each node of the graph.



Example



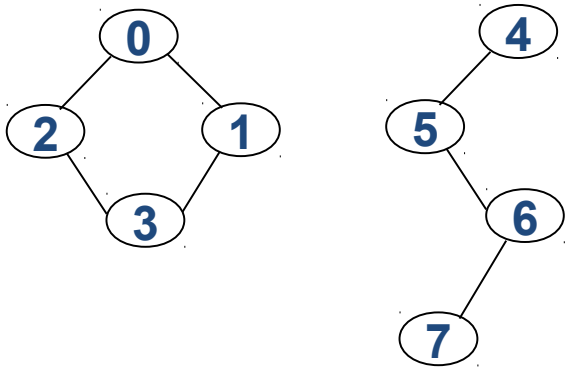
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

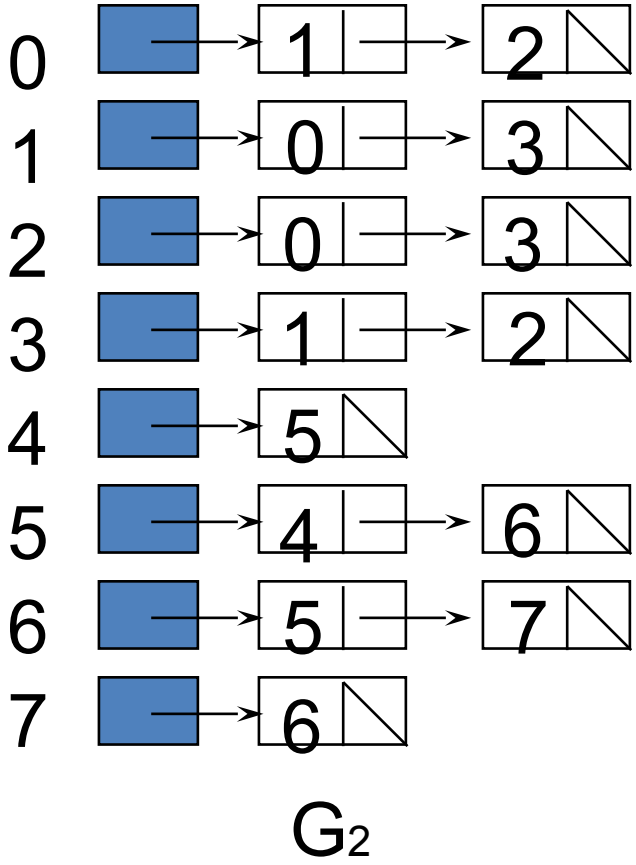


G_1

Undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

Example

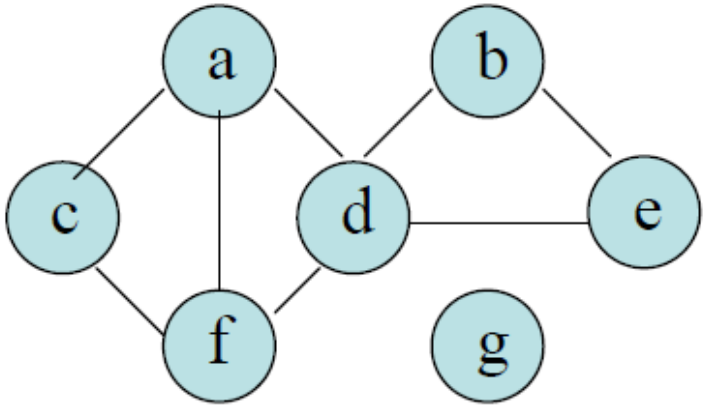




G_2

Graph Representation

- Incidence Matrix



	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

Graph Implementations

Consideration: Suppose that the number of nodes in the graph is constant; arcs may be added or deleted but nodes may not.

DECLARATION:

```
#define MAXNODES 50
struct node{
  /* information associated with each node*/
};
struct arc{
  int adj;
  /* information associated with each arc*/
};
```

The value of g.arcs[i][j].adj is either TRUE(1) or FALSE(0).

Graph Implementations

```
struct graph{
  struct node nodes[MAXNODES];
  struct arc arcs[MAXNODES][MAXNODES];
};
struct graph g;
```

Function: Join

```
void join(int adj[][MAXNODES],int node1,int node2) {
  /* add an arc from node1 to node2*/
 adj[node1][node2]=TRUE;
};//end join
```

Function: Remove

```
void remv(int adj[][MAXNODES],int node1,int node2) {
  /* delete arc from node1 to node2 if one exists*/
 adj[node1][node2]=FALSE;
};//end remv
```

Example

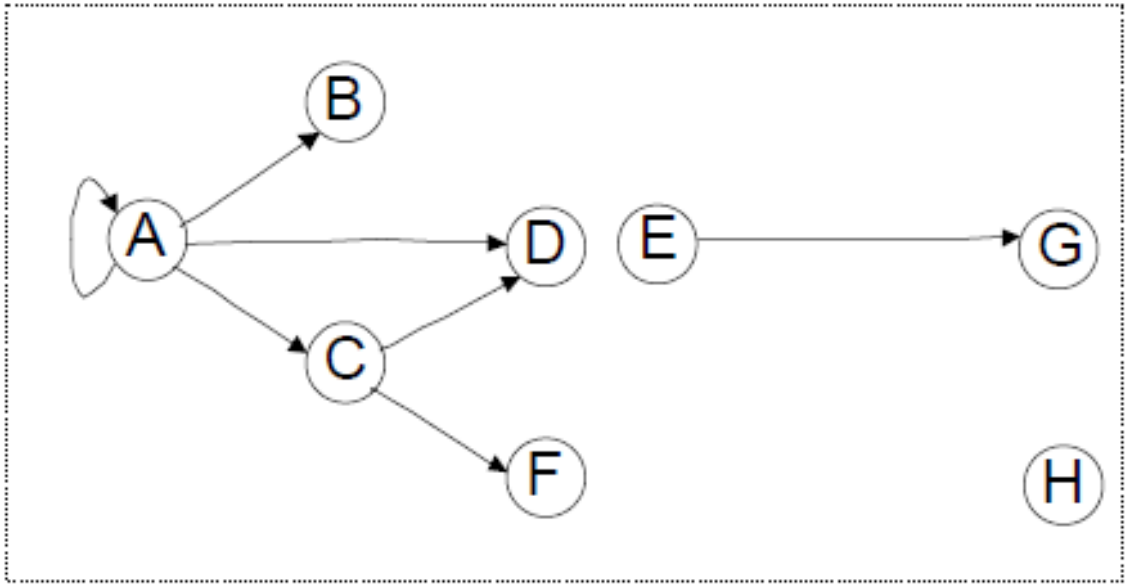
A weighted graph with a fixed number of nodes may be declared by

```
struct arc{
        int adj;
        int weight;
};
struct arc g[MAXNODES][MAXNODES];

void joinwt(struct arc g[ ][MAXNODES], int node1,
                        int node2, int wt){
g[node1][node2].adj=TRUE;
g[node1][node2].weight=wt;
}//end joinwt
```

Drill

1. For the given graph, determine the following:
   a. Find its adjacency matrix
   b. Find its incidence matrix
   c. Represent the adjacency list using linked structure



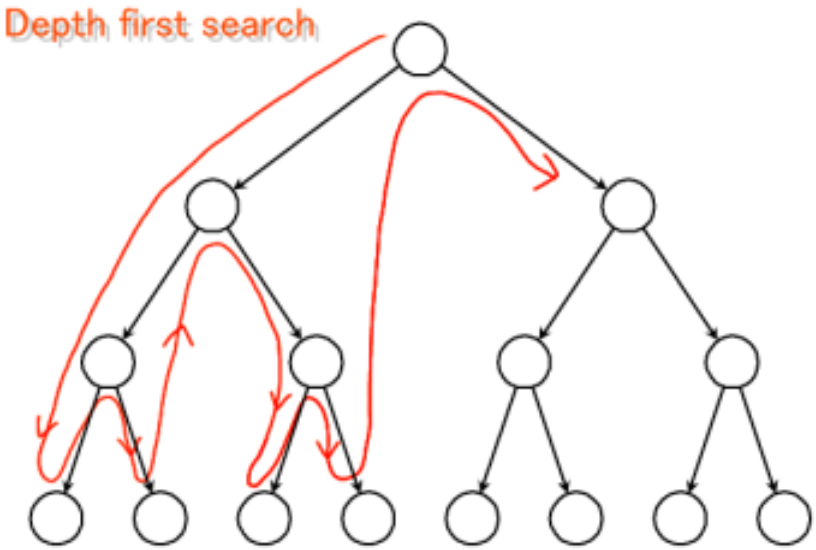
Drill

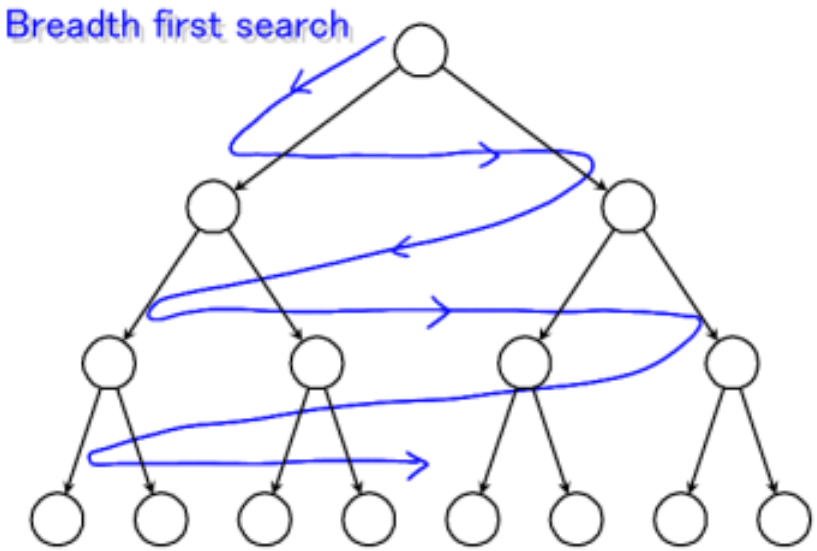
2. Draw a digraph that will represent to each of the following relations on the integers from 1 to 12:

   a. x is related to y if x-y is evenly divisible by 3.

   b. x is related to y if the remainder on division of x by y is 2.

Graph Traversals





- Both take time: O(V+E)

Recall: Transitive closure

- Consider the relation R = { (1, 3), (1, 4), (2, 1) , (3, 2)} on the set { 1, 2, 3, 4}. This relation is not transitive because it does not contain all pairs of the form (a , c) where (a , b) and (b, c) are in R.
- Add (1, 2), (2, 3), (2, 4), and (3, 1).

Transitive closure of R

- Let R be a relation on a set A with n elements. Then,

$$\text{transitive}(R) = R \cup R^2 \cup \ldots \cup R^n$$

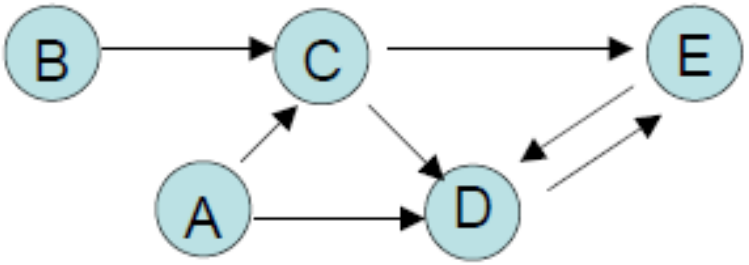
If the graph has *n* nodes, it must be true that

$$\text{path}[i][j] == \text{adj}[i][j] \;||\; \text{adj}_2[i][j] \;||\; \cdots \text{adj}_n[i][j]$$

There must be a path from i to j of length less than or equal to *n* (no of nodes).

Transitive closure

Example:



Adjacency Matrices:

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

(a) adj

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

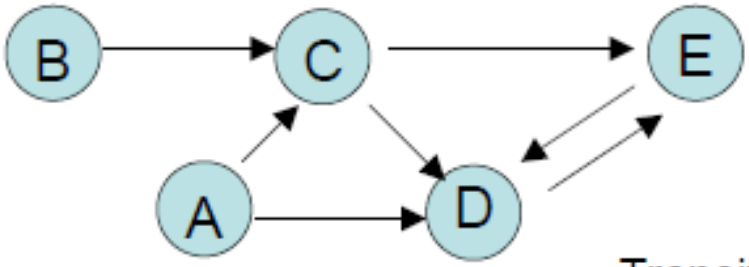
(b) adj_2

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

(c) adj_3

Transitive closure

Example:



Transitive Closure Matrix Path:

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

(d) adj_4

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

(e) adj_5

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

Path = adj or adj_2 or adj_3 or adj_4 or adj_5

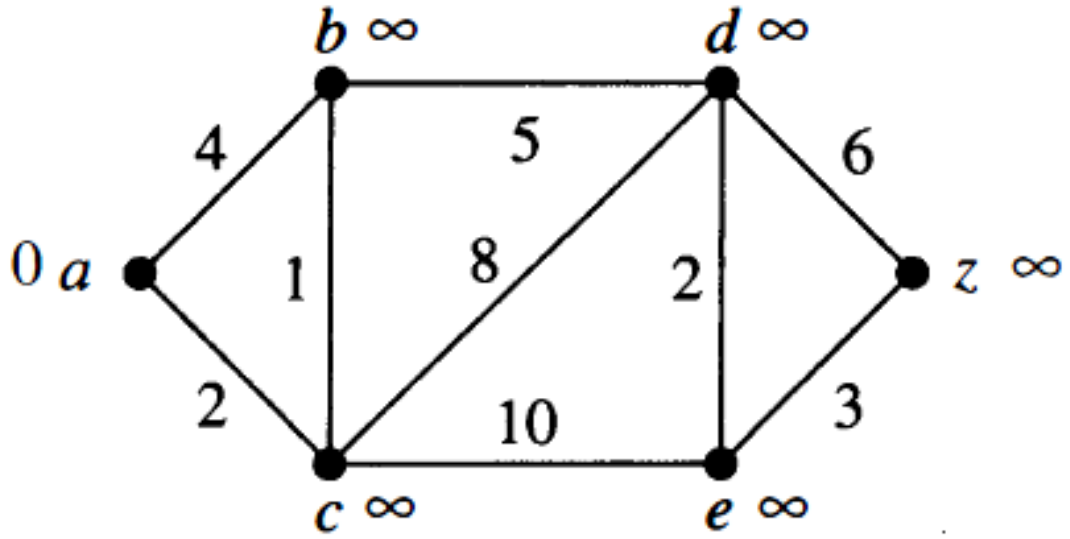
Shortest Path Problem

Finding shortest path is a classical problem in graph theory. Edges are assigned certain weights representing, for example, distances between cities, times separating the execution of certain tasks, cost of transmitting information between locations, amounts of some substance transported from one place to another.

DIJKSTRA'S ALGORITHM

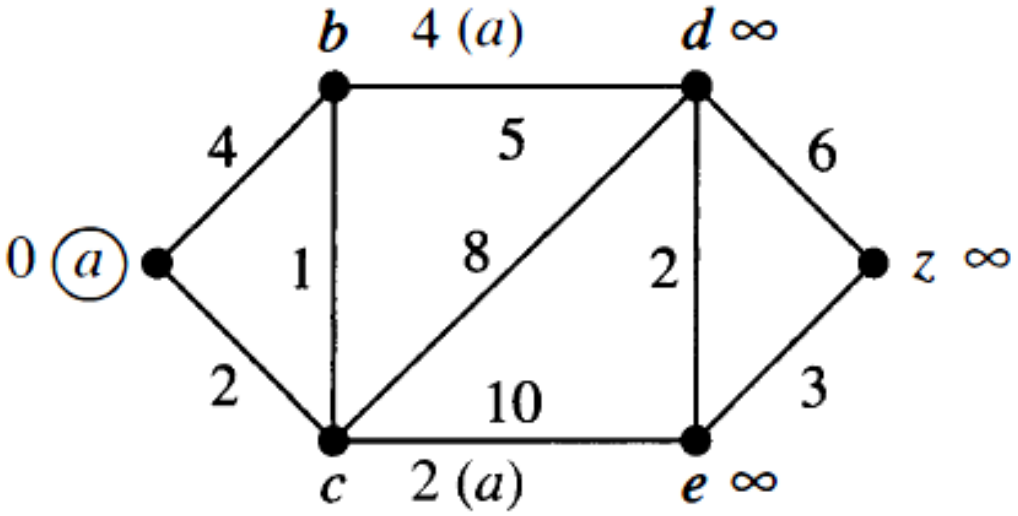
A number of paths P_1.. P_n from a vertex v are tried, and each time, the shortest path is chosen among them, which may mean that the same path P_i can be continued by adding one more edge to it. But if P_i turns out to be longer than any other path that can be tried, P_i is abandoned and this other path is tried by resuming from where it was left and by adding one more edge to it.

Example



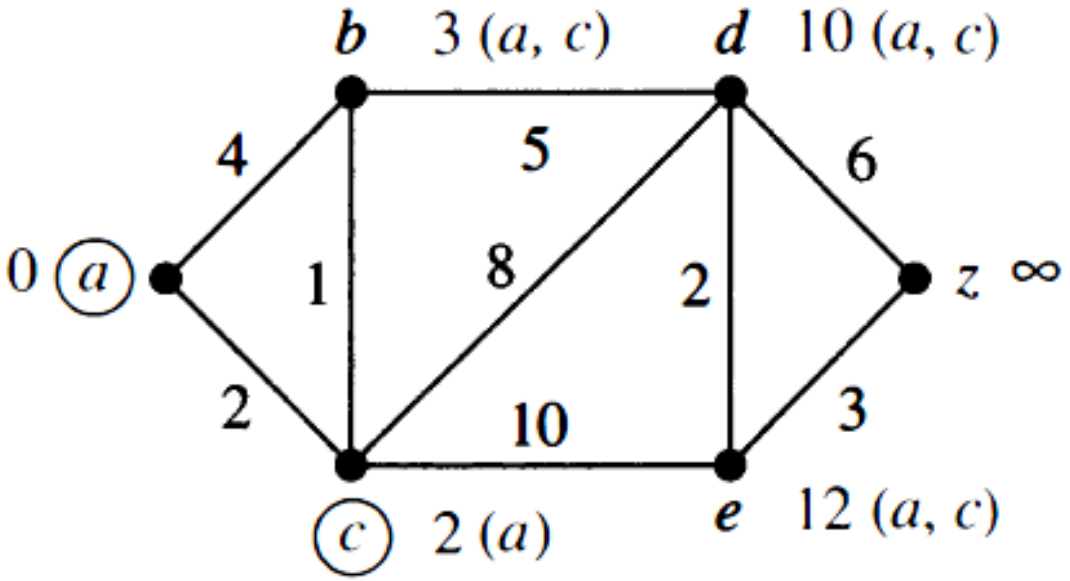
(a)

Example



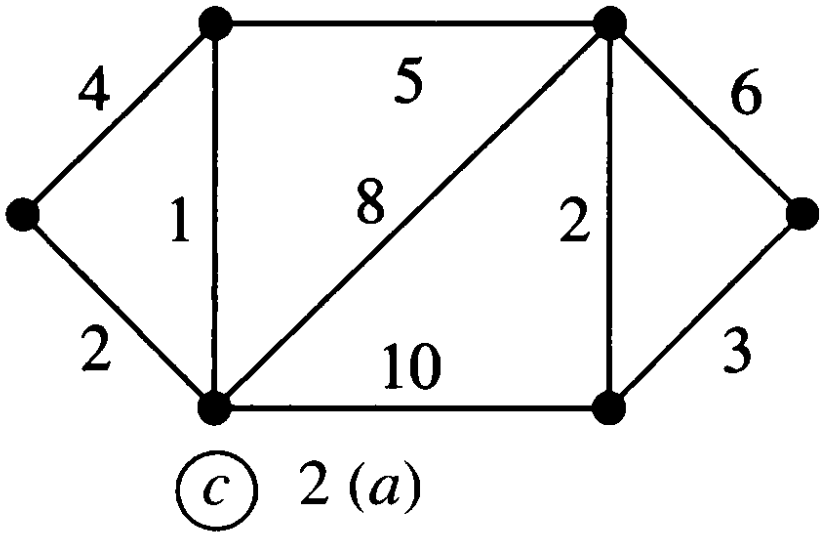
(b)

Example



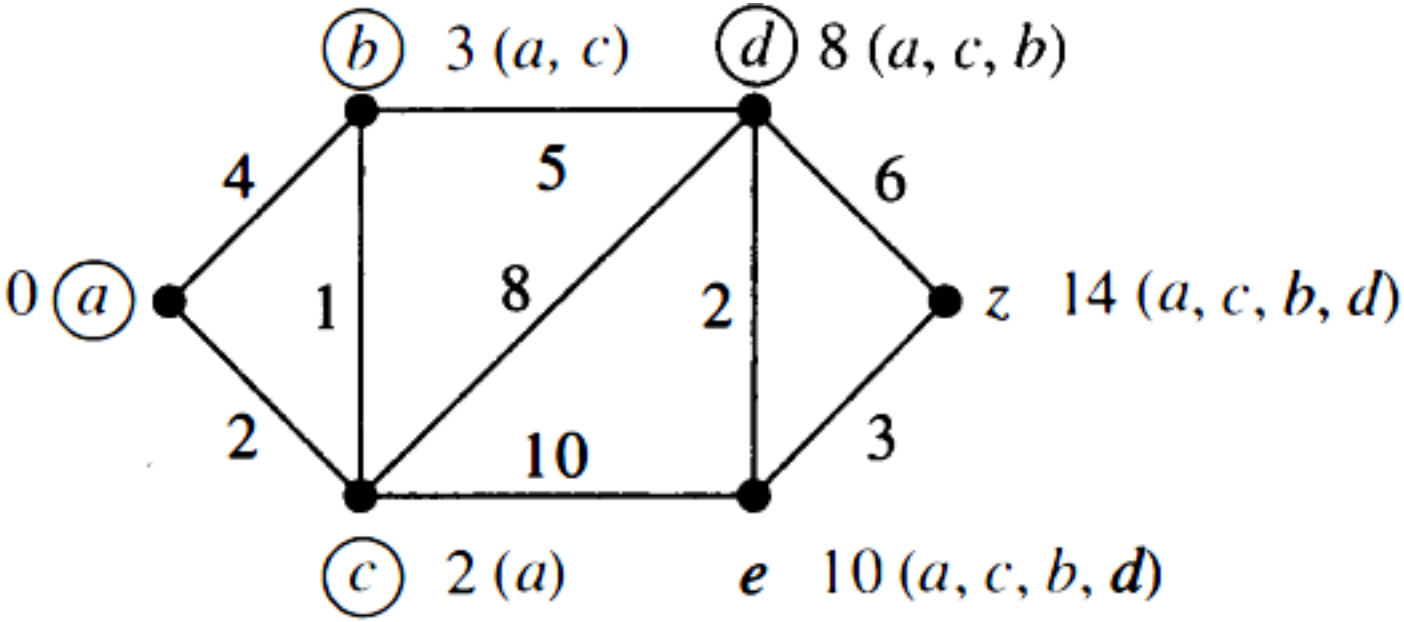
(c)

Example



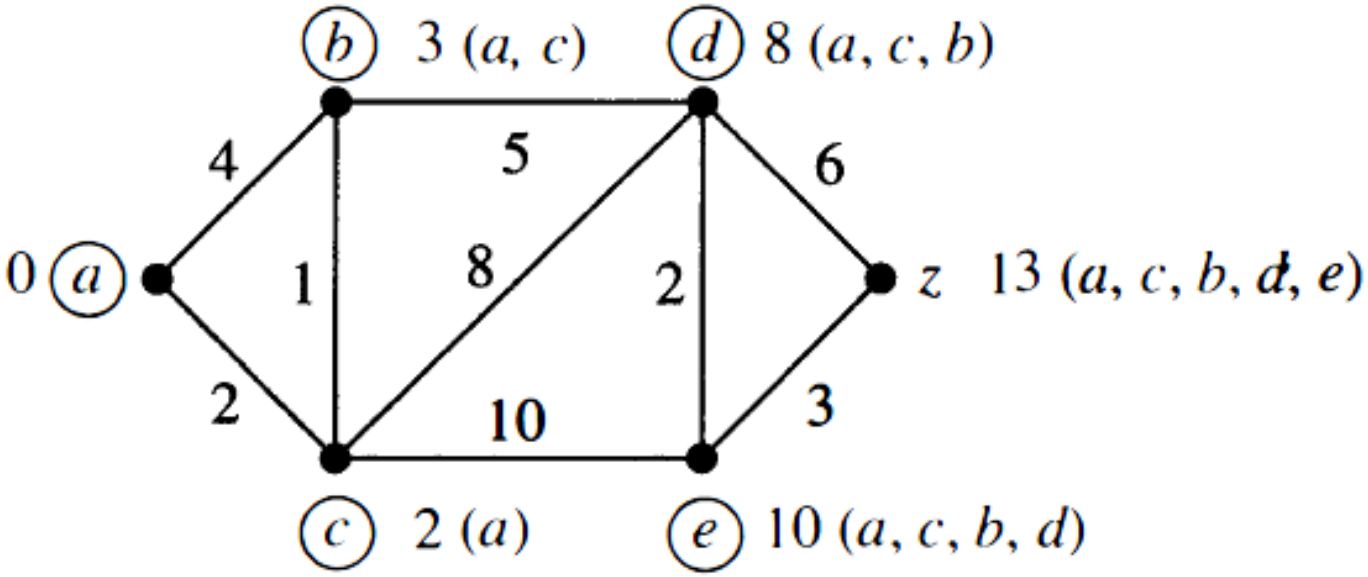
(d)

Example



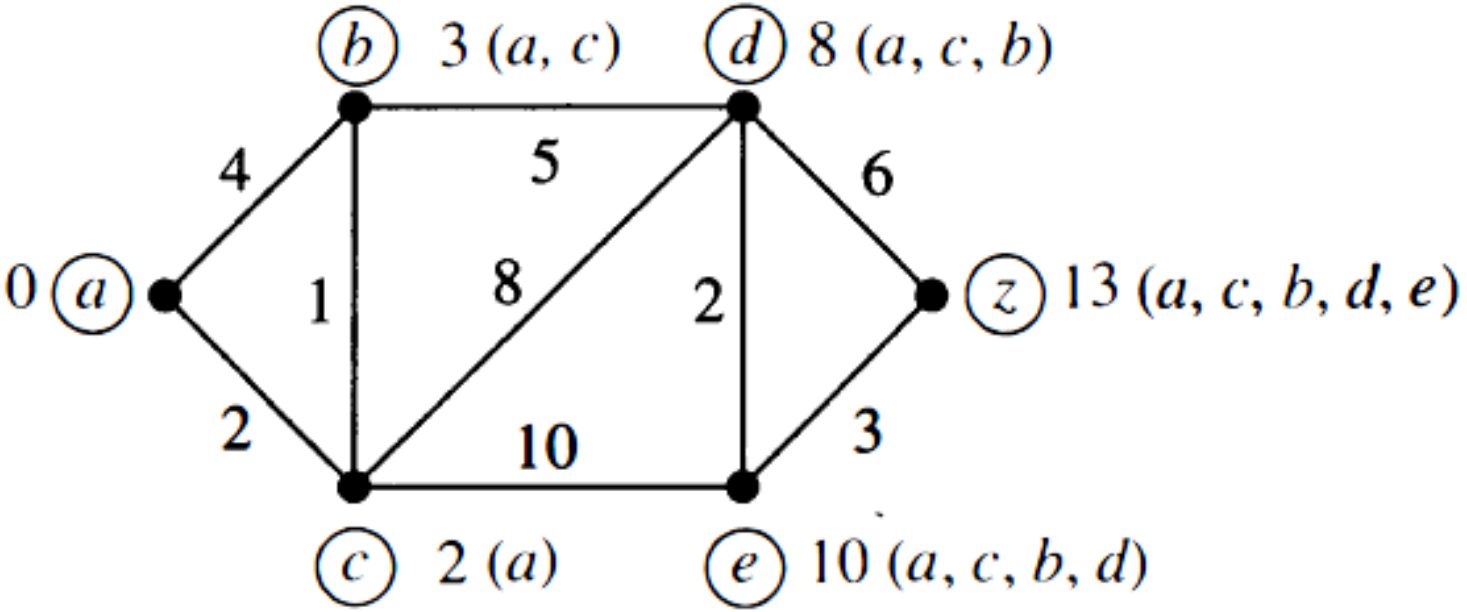
(e)

Example



(f)

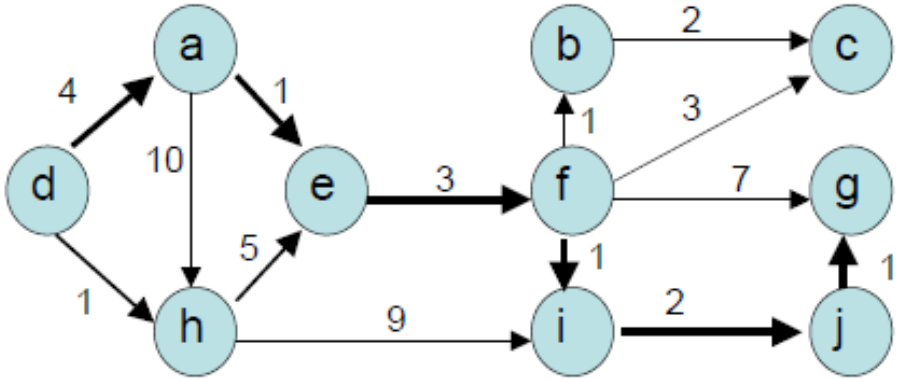
Example



(g)

```
DijkstraAlgorithm(weighted simple Digraph,  vertex first)
for all vertices v
  currDist(v) = α;
currDist(first)= 0;
toBeChecked= all vertices;
while toBeChecked is not empty
    v = a vertex in toBeChecked with minimal currDist(v);
   remove v from toBeChecked;
   for all vertices u adjacent to v and in toBeChecked
       if currDist(u) >currDist(v) + weight(edge(vu))
          currDist(u) = currDist(v) + weight (edge(vu));
          predecessor(u)=v;
```

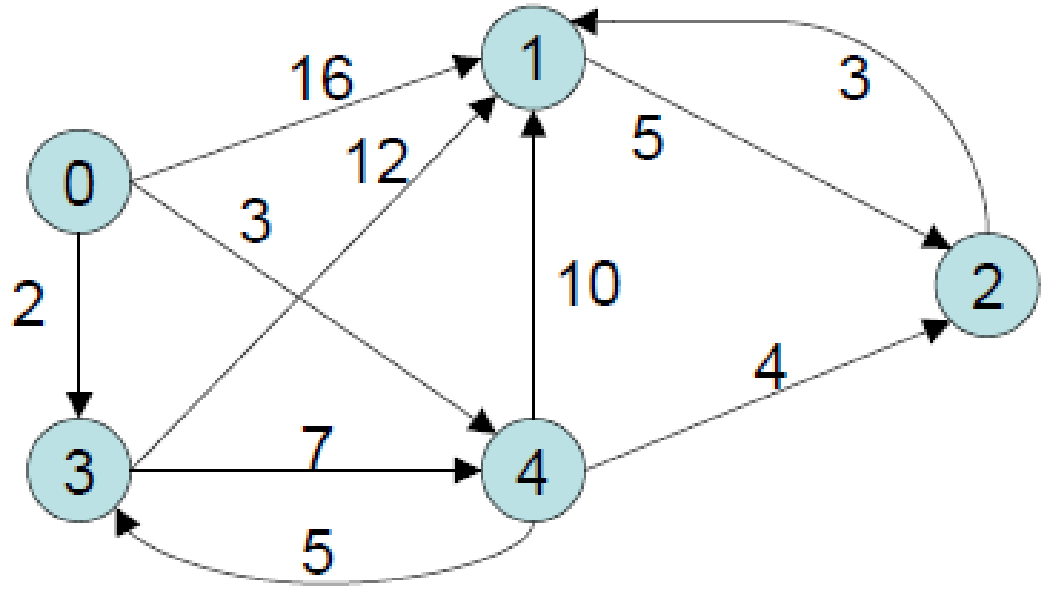
Drill



Iteration:	init	1	2	3	4	5	6	7	8	9	10
Active vertex:		d	h	a	e	f	b	i	c	j	g
a	α	4	4								
b	α	α	α	α	α	9					
c	α	α	α	α	α	11	11	11			
d	0										
e	α	α	6	5							
f	α	α	α	α	8						
g	α	α	α	α	α	15	15	15	15	12	12
h	α	1									
i	α	α	10	10	10	9	9				
j	α	α	α	α	α	α	α	11	11		

Exercise:

1. Consider the given graph. Find the shortest distance from node 0 to every other node in the graph.



End