Unit 5
Graphs

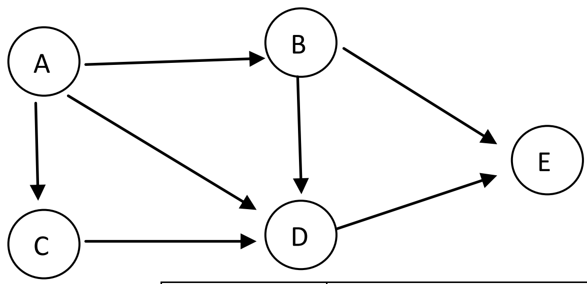
Explain topological sort with suitable algorithm and example (16) (MAY/JUNE 2012)

Definition: A **topological sort** is a linear ordering of vertices in a directed acyclic graph such that if there is a path from V_i to V_j, then V_j appears after V_i in the linear ordering.

Topological ordering is not possible. If the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.
To implement the topological sort, perform the following steps.
Step 1 : - Find the indegree for every vertex.
Step 2 : - Place the vertices whose indegree is `0' on the empty queue.
Step 3 : - Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.
Step 4 : - Enqueue the vertex on the queue, if its indegree falls to zero.
Step 5 : - Repeat from step 3 until the queue becomes empty.
Step 6 : - The topological ordering is the order in which the vertices dequeued.
Routine to perform Topological Sort

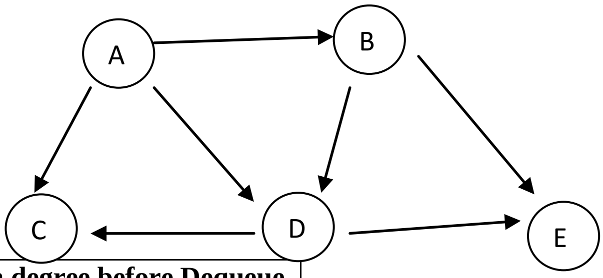
```
void topsort (graph g)
{
queue q ;
int counter = 0;
vertex v, w ;
q = createqueue (numvertex);
makeempty (q);
for each vertex v
if (indegree [v] = = 0)
enqueue (v, q);
while (! isempty (q))
{
v = dequeue (q);
topnum [v] = + + counter;
for each w adjacent to v
if (--indegree [w] = = 0)
enqueue (w, q);
}
if (counter ! = numvertex)
error (" graph has a cycle");
disposequeue (q); /* free the memory */
}
```

Example 1:



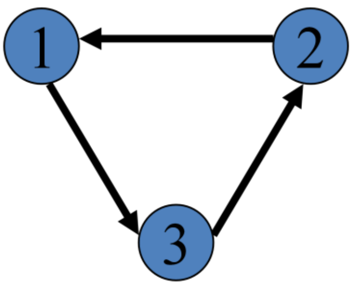
Vertex	In degree before Dequeue				
	1	2	3	4	5
A	**0**	**0**	**0**	**0**	**0**
B	**1**	**0**	**0**	**0**	**0**
C	**1**	**0**	**0**	**0**	**0**
D	**3**	**2**	**1**	**0**	**0**
E	**2**	**2**	**1**	**1**	**0**
Enqueue	**A**	**B,C**		**D**	**E**
Dequeue	**A**	**B**	**C**	**D**	**E**
Dequeue	**A**	**C**	**B**	**D**	**E**

Example 2:



Vertex	In degree before Dequeue				
	1	2	3	4	5
A	0	0	0	0	0
B	1	0	0	0	0
C	2	1	1	0	0
D	2	1	0	0	0
E	2	2	1	0	0
Enqueue	A	B	D	C,E	-
Dequeue	A	B	D	C	E
Dequeue	A	B	D	E	C

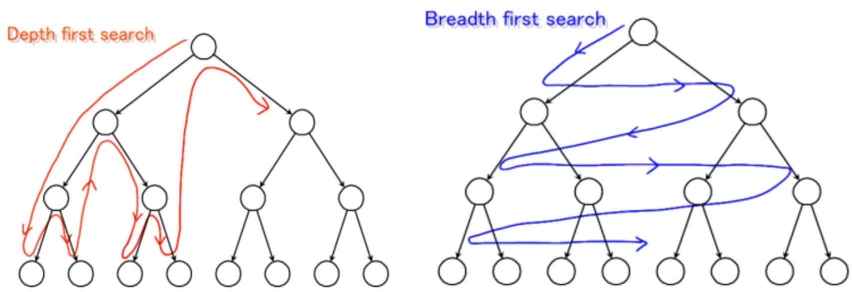
Example 3:



Error

Graph Traversals

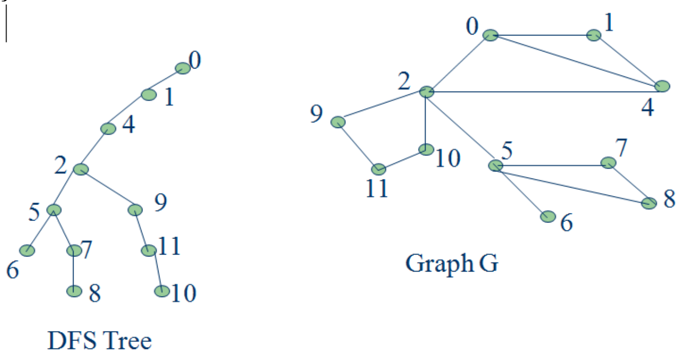
1. BFS
2. DFS



DFS follows the following rules:

1. Select an unvisited node x, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
4. Repeat steps 3 and 4 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from step 1.

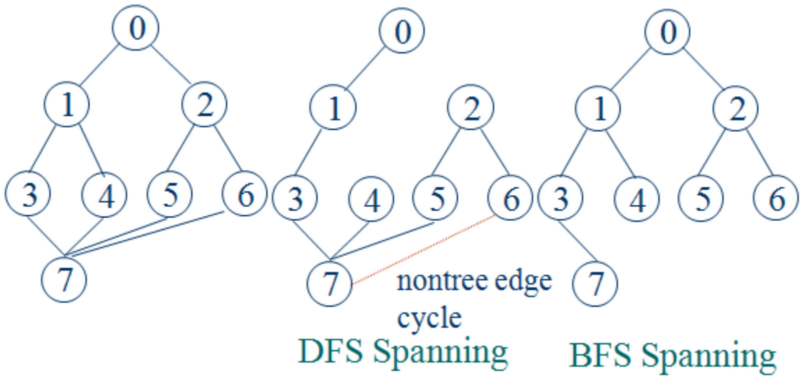
```
DFS(input: Graph G) {
        Stack S; Integer x, t;
        while (G has an unvisited node x){
                visit(x); push(x,S);
                while (S is not empty){
                        t := peek(S);
                        if (t has an unvisited neighbor y){
        visit(y); push(y,S); }
                        else
                                pop(S);
                }
        }
}
```

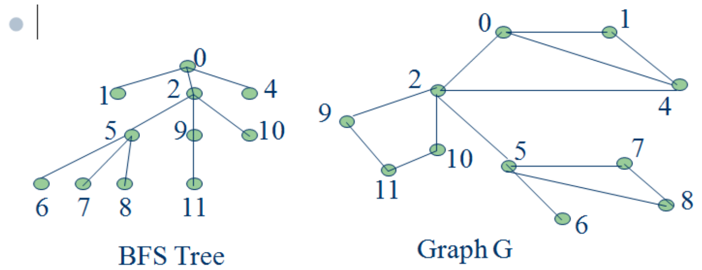


BFS follows the following rules:

1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
3. Repeat step 2 until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from Step 1.

```
BFS(input: graph G) {
        Queue Q;   Integer x, z, y;
        while (G has an unvisited node x) {
                visit(x); Enqueue(x,Q);
                while (Q is not empty){
                        z := Dequeue(Q);
                        for all (unvisited neighbor y of z){
        visit(y); Enqueue(y,Q);
                        }
                }
        }
}
```





Minimum Spanning Tree Algorithm:

i. **Prim's algorithm**
ii. **Kruskals algorithm**

Write and explain the prim's algorithm with an example (16) (MAY/JUNE 2012) (NOV/DEC 2011)

Prim's Algorithm to Construct a Minimal Spanning Tree
Input: A weighted, connected and undirected graph G = (V,E).
Output: A minimal spanning tree of G.
Step 1: Let x be any vertex in V. Let X = and Y = V
Step 2: Select an edge (u,v) from E such that, and (u,v)has the smallest weight among edges between X and Y.
Step 3: Connect u to v.
Step 4: If Y is empty, terminate and the resulting tree is a minimal spanning tree. Otherwise, go to Step 2.

Initialization

a. Pick a vertex ***r*** to be the root
b. Set $D(r) = 0$, $parent(r) = null$
c. For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$
d. Insert all vertices into priority queue ***P***,
using distances as the keys

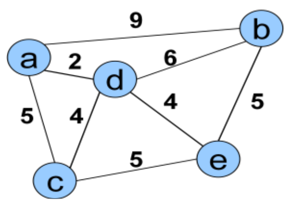
While *P* is not empty

{

1. Select the next vertex ***u*** to add to the tree
$u = P.deleteMin()$
2. Update the weight of each vertex ***w*** adjacent to ***u*** which is **not** in the tree (i.e., $w \in P$)
If $weight(u,w) < D(w)$,
a. $parent(w) = u$
b. $D(w) = weight(u,w)$
c. Update the priority queue to reflect new distance for ***w***

}

Eg.1



Initial matrix

Node	cost	visited	Parent Node
a	**0**	0	-
b	∞	0	-
c	∞	0	-
d	∞	0	-
e	∞	0	-

Node a is visited

Node	cost	visited	Parent Node
a	**0**	1	-
b	Min(∞,9)	0	a
c	Min(∞,5)	0	a
d	**Min(∞,2)**	0	a
e	∞	0	-

Node	cost	visited	Parent Node
a	**0**	1	-
b	9	0	a
c	5	0	a
d	2	0	a
e	∞	0	-

Node d is visited

Node	cost	visited	Parent Node
a	**Min(0,2)**	1	-,d
b	Min(9,6)	0	a,d
c	Min(5,4)	0	a,d
d	2	1	a
e	Min(∞,4)	0	-,d

Node c is visited

Node	cost	visited	Parent Node
a	Min(0,5)	1	-,c
b	6	0	d
c	4	1	d
d	Min(2,4)	1	a,c
e	Min(4,5)	0	d,c

Node	cost	visited	Parent Node
a	0	1	-
b	6	0	d
c	4	1	d
d	2	1	a
e	4	0	d

Node e is visited

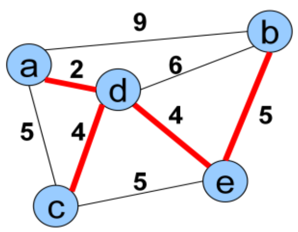
Node	cost	visited	Parent Node
a	0	1	-
b	min(6,5)	0	d,e
c	min(4,5)	1	d,e
d	min(2,4)	1	a,e
e	4	1	d

Node	cost	visited	Parent Node
a	0	1	-
b	5	0	d,e
c	4	1	d,e
d	2	1	a,e
e	4	1	d

Node b is visited

Node	cost	visited	Parent Node
a	Min(0,9)	1	-,b
b	5	1	e
c	4	1	d
d	Min(2,6)	1	a,b
e	Min(4,5)	1	D,b

Node	cost	visited	Parent Node
a	0	1	-
b	5	1	E
c	4	1	d
d	2	1	a
e	4	1	d



Cost = 2+4+4+5 = 15

Explain Kruskal's algorithm with an example (16)

Kruskal's Algorithm to Construct a Minimal Spanning Tree

Input: A weighted, connected and undirected graph $G = (V,E)$.

Output: A minimal spanning tree of G.

Step 1: $T = \Phi$

Step 2: while T contains less than n-1edges **do**

Choose an edge (v,w) from E of the smallest weight.

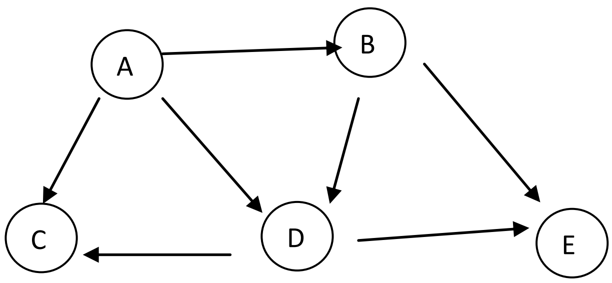
Delete (v,w) from E.

If the adding of (v,w)does not create cycle in T **then**

Add (v,w) to T.

Else Discard (v,w).

end while



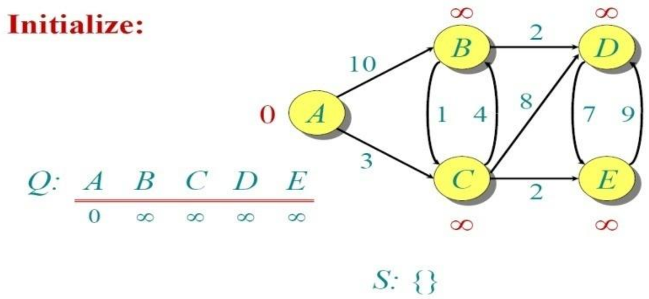
Edge	Weight	Action	Comments
(A,F)	10	Accepted	An edge with minimum cost(from delete_min of heap
(C,D)	12	Accepted	
(B,G)	14	Accepted	
(B,C)	16	Accepted	
(D,E)	22	Accepted	
(E,G)	24	Rejected	Forms a cycle
(F,E)	25	Accepted	
(A,B)	28	Rejected	Forms a cycle

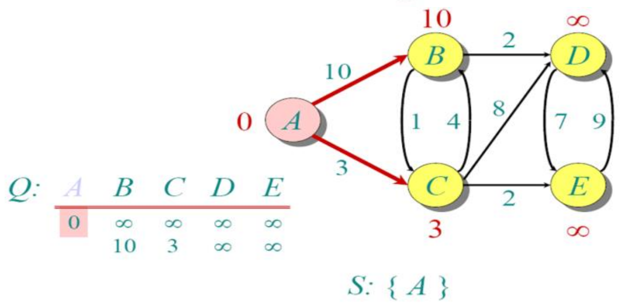
1. Single-source shortest paths in weighted graphs
   i. Dijkstra's Algorithm
   ii. Bellman-Ford Algorithm
2. All pair shortest path algorithm
   a. Floyd–Warshall Algorithm

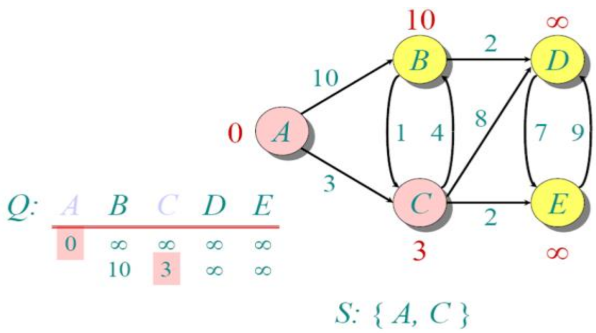
Dijkstra's Algorithm

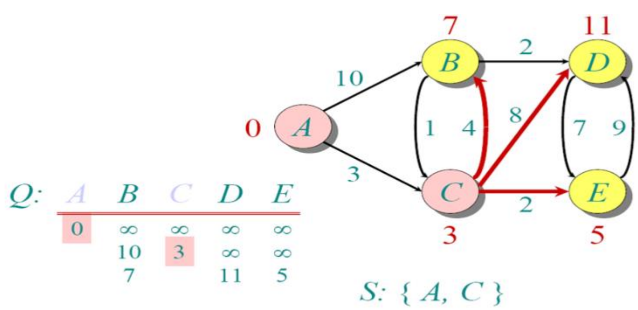
```
Dijkstra(G,s)
for each vertex u Î G.V()
   u.setd(¥)
   u.setparent(NIL)
s.setd(0)
S ¬ Æ
Q.init(G.V())
while not Q.isEmpty()
   u ¬ Q.extractMin()
```

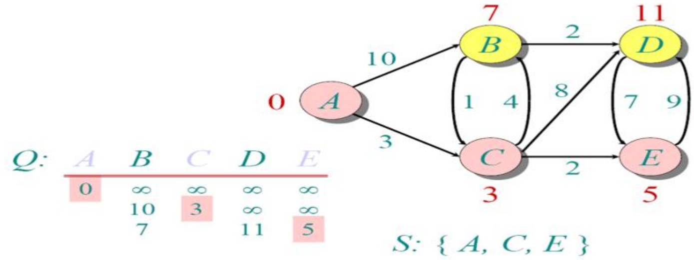
```
S ¬ S È {u}
for each v Î u.adjacent() do
  Relax(u, v, G)
  Q.modifyKey(v)

Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
 v.setd(u.d()+G.w(u,v))
v.setparent(u)
```

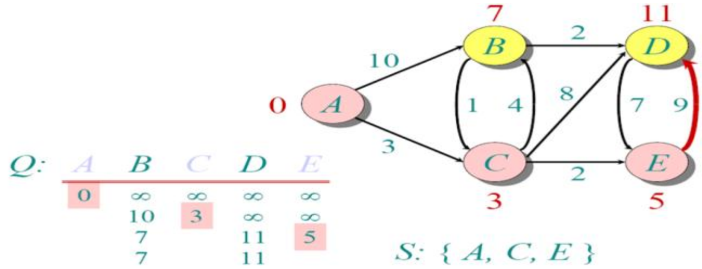


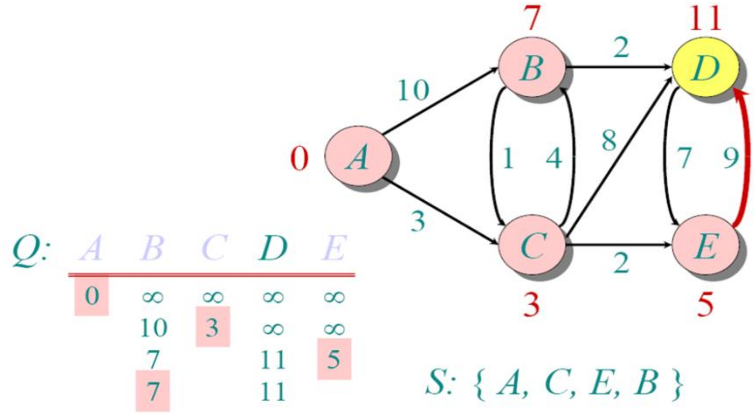


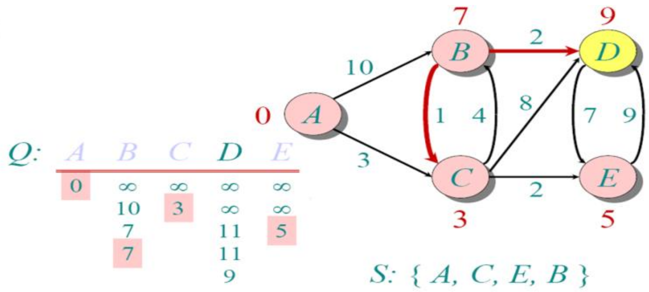


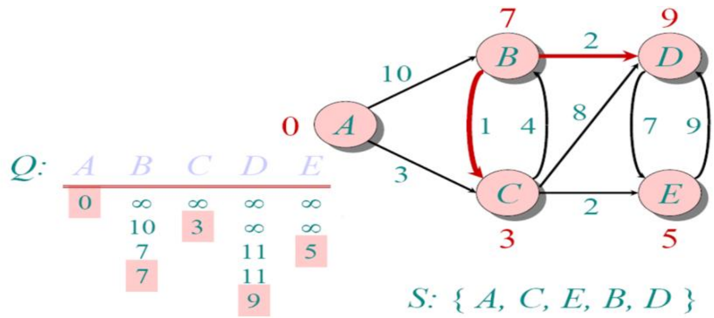



Q: A B C D E
0 ∞ ∞ ∞ ∞
10 3 ∞ ∞
7 11 5
S: { A, C, E }


Q: A B C D E
0 ∞ ∞ ∞ ∞
10 3 ∞ ∞
7 11 5
7 11
S: { A, C, E }


Q: A B C D E
0 ∞ ∞ ∞ ∞
10 3 ∞ ∞
7 11 5
7 11
S: { A, C, E, B }


Q: A B C D E
0 ∞ ∞ ∞ ∞
10 3 ∞ ∞
7 11 5
7 11
9
S: { A, C, E, B }


Q: A B C D E
0 ∞ ∞ ∞ ∞
10 3 ∞ ∞
7 11 5
7 11
9
S: { A, C, E, B, D }

Initial matrix

Node	cost	visited	Parent Node
a	**0**	0	-
b	∞	0	-
c	∞	0	-
d	∞	0	-
e	∞	0	-

Node a is visited

Node	cost	visited	Parent Node
a	**0**	1	-
b	10	0	a
c	**3**	0	a
d	∞	0	-
e	∞	0	-

Node c is visited

Node	cost	visited	Parent Node
a	**0**	1	-
b	10,(3+4)	0	a,c
c	**3**	1	a
d	∞,(3+8)	0	c
e	∞,**(3+2)**	0	-,c

Node e is visited

Node	cost	visited	Parent Node
a	**0**	1	-
b	7	0	c
c	**3**	1	a
d	11,(5+9)	0	c,e
e	**5**	1	c

Node b is visited

Node	cost	visited	Parent Node
a	**0**	1	-
b	7	1	c
c	**3,(7+1)**	1	a,b
d	11,(7+2)	0	c,b
e	**5**	1	c

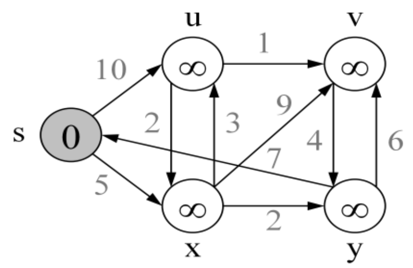
Node d is visited

Node	cost	visited	Parent Node
a	**0**	1	-
b	7	1	c
c	**3**	1	a
d	9	1	b
e	**5,(9+7)**	1	C,d

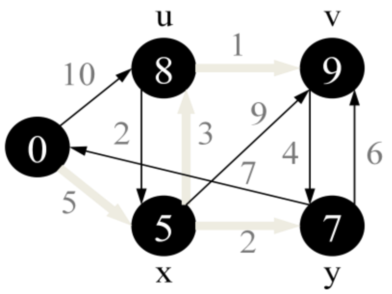
Final matrix

Node	cost	visited	Parent Node
a	**0**	1	-
b	**7**	1	c
c	**3**	1	a
d	**9**	1	b
e	**5**	1	c

Example .2



Solution:



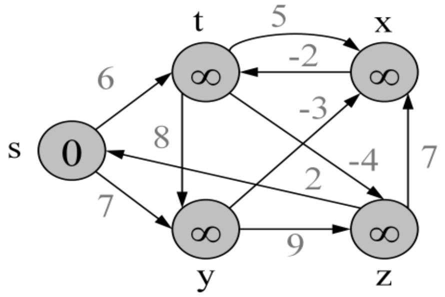
- Dijkstra’s doesn’t work when there are negative edges:
  - Intuition – we can not be greedy any more on the assumption that the lengths of paths will only increase in the future
- Bellman-Ford algorithm detects negative cycles (returns *false*) or returns the shortest path-tree

Bellman-Ford algorithm

```
Bellman-Ford(G,s)
for each vertex u Î G.V()
   u.setd(¥)
   u.setparent(NIL)
s.setd(0)
for i ¬ 1 to |G.V()|-1 do
    for each edge (u,v) Î G.E() do
             Relax (u,v,G)
for each edge (u,v) Î G.E() do
          if v.d() > u.d() + G.w(u,v) then
```

```
      return false
return true

Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
 v.setd(u.d()+G.w(u,v))
v.setparent(u)
```

Eg.1



Node	cost	visited	Parent Node
s	0	0	-
t	∞	0	-
x	∞	0	-
y	∞	0	-
z	∞	0	-

Node S is Visited

Node	cost	visited	Parent Node
s	0	1	-
t	6	0	s
x	∞	0	-
y	7	0	s
z	∞	0	-

Node t is Visited (-ve cost)

Node	cost	visited	Parent Node
s	0	1	-
t	6	1	s
x	∞	0	-
y	7	0	s
z	(6-4)	0	t

Node y is Visited (-ve cost)

Node	cost	visited	Parent Node
s	0	1	-
t	6	1	s
x	∞,(7-3)	0	-,y
y	7	1	s
z	2	0	t

Node	cost	visited	Parent Node
s	0	1	-
t	6	1	s
x	4	0	y
y	7	1	s
z	2	0	t

Node x is Visited (-ve cost)

Node	cost	visited	Parent Node
s	0	1	-
t	Min(6,(4-2))	1	s,x
x	4	1	y
y	7	1	s
z	2	0	t

Node	cost	visited	Parent Node
s	0	1	-
t	2	1	x
x	4	1	y
y	7	1	s
z	2	0	t

Update t

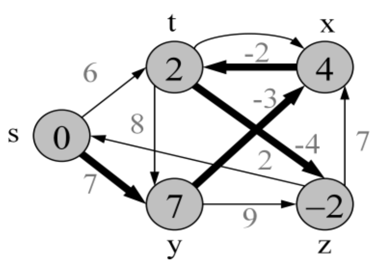
Node	cost	visited	Parent Node
s	0	1	-
t	2	1	x
x	4	1	y
y	7	1	s
z	Min(2,(2-4))	0	t

Final matrix

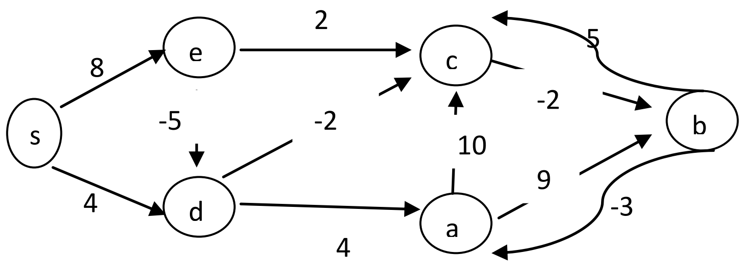
Node	cost	visited	Parent Node
s	0	1	-
t	2	1	x
x	4	1	y
y	7	1	s
z	-2	1	t

The cost of moving from

s to s is 0
S to t is 2
S to x is 4
S to y is 7
S to z is -2.



Eg.2



Initial matrix

Node	cost	visited	Parent Node
s	0	0	-
a	∞	0	-
b	∞	0	-
c	∞	0	-
d	∞	0	-
e	∞	0	-

Node s is visited

Node	cost	visited	Parent Node
s	0	1	-
a	∞	0	-
b	∞	0	-
c	∞	0	-
d	4	0	s
e	8	0	s

Node e is visited

Node	cost	visited	Parent Node
s	0	1	-
a	∞	0	-
b	∞	0	-
c	∞	0	-
d	4,(8-5)	0	s,e
e	8	1	s

Node	cost	visited	Parent Node
s	0	1	-
a	∞	0	-
b	∞	0	-
c	∞	0	-
d	3	0	e
e	8	1	s

Node d is visited

Node	cost	visited	Parent Node
s	0	1	-
a	∞	0	-
b	∞	0	-
c	∞,(3-2)	0	-,d
d	3	1	e
e	8	1	s

Node	cost	visited	Parent Node
s	0	1	-
a	∞	0	-
b	∞	0	-
c	1	0	-,d
d	3	1	e
e	8	1	s

Node c is visited

Node	cost	visited	Parent Node
s	0	1	-
a	∞	0	-
b	∞,(1-2)	0	-,c
c	1	1	-,d
d	3	1	e
e	8	1	s

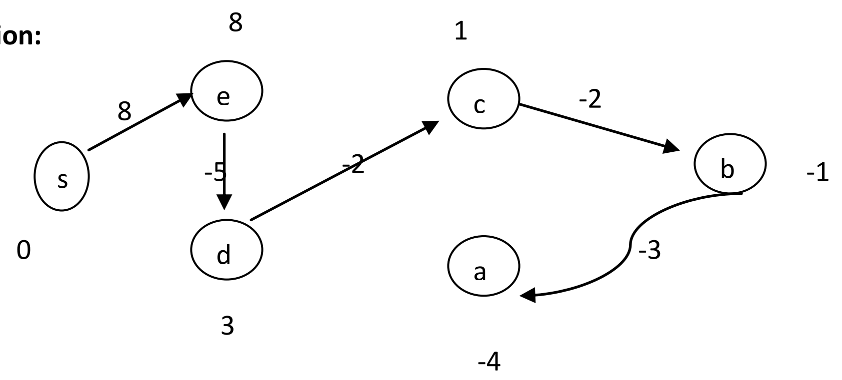
Node b is visited

Node	cost	visited	Parent Node
s	0	1	-
a	∞,(-1-3)	0	-,b
b	-1	1	c
c	1	1	d
d	3	1	e
e	8	1	s

Node a is visited, but no –ve values.

Node	cost	visited	Parent Node
s	0	1	-
a	-4	1	b
b	-1	1	c
c	1	1	d
d	3	1	e
e	8	1	s

Solution:

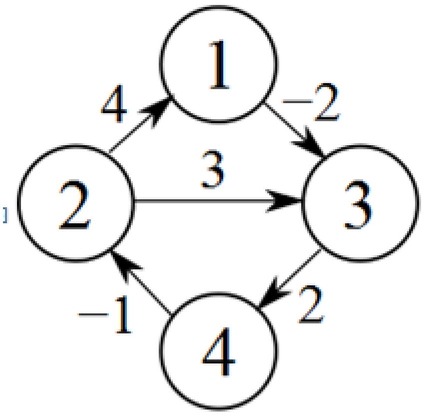


Floyd–Warshall Algorithm

FloydWarshall(**matrix** W, **integer** *n*)
for *k* ← 1 **to** *n* **do**
for *i* ← 1 **to** *n* **do**
for *j* ← 1 **to** *n* **do**
$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
// **D[i,j] = min(D[i,j],D[i,k]+D[k,j])**
return $D^{(n)}$

The final D matrix will store all the shortest paths.
Adjacency matrix for the given graph.

	1	**2**	**3**	**4**
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	-1	∞	0



K=0

K	i	j	d[i,j]	d[i,k]	d[k,j])	d[i,k]+ d[k,j]	d[i,j]	Min(d[i,j], d[i,k]+ d[k,j])
1	1	1	0	0	0	0	0	0
1	1	2	∞	0	∞	∞	∞	∞
1	1	3	-2	0	-2	-2	-2	-2
1	1	4	∞	0	∞	∞	∞	∞
1	2	1	4	4	0	4	4	4
1	2	2	0	4	∞	∞	0	0
1	2	3	3	4	-2	2	3	2
1	2	4	∞	4	∞	∞	∞	∞
1	3	1	∞	∞	0	∞	∞	∞
1	3	2	∞	∞	∞	∞	∞	∞
1	3	3	0	∞	-2	∞	0	0
1	3	4	2	∞	∞	∞	2	2
1	4	1	∞	∞	0	∞	∞	∞
1	4	2	-1	∞	∞	∞	-1	-1
1	4	3	∞	∞	-2	∞	∞	∞
1	4	4	0	∞	∞	∞	0	0

K=1

	1	**2**	**3**	**4**
1	0	∞	-2	∞
2	4	0	**2**	∞
3	∞	∞	0	2
4	∞	-1	∞	0

k	i	j	d[i,j]	d[i,k]	d[k,j]	d[i,k]+ d[k,j]	d[i,j]	Min(d[i,j], d[i,k]+ d[k,j])
2	1	1	0	∞	4	∞	0	0
2	1	2	∞	∞	0	∞	∞	∞
2	1	3	-2	∞	2	∞	-2	-2
2	1	4	∞	∞	∞	∞	∞	∞
2	2	1	4	0	4	4	4	4
2	2	2	0	0	0	0	0	0
2	2	3	2	0	2	2	2	2
2	2	4	∞	0	∞	∞	∞	∞
2	3	1	∞	∞	4	∞	∞	∞
2	3	2	∞	∞	0	∞	∞	∞
2	3	3	0	∞	2	∞	0	0
2	3	4	2	∞	∞	∞	2	2
2	4	1	∞	-1	4	3	∞	3
2	4	2	-1	-1	0	-1	-1	-1
2	4	3	∞	-1	2	1	∞	1
2	4	4	0	-1	∞	∞	0	0

K=2

	1	**2**	**3**	**4**
1	0	∞	-2	∞
2	4	0	**2**	∞
3	∞	∞	0	2
4	3	-1	1	0

k	i	j	d[i,j]	d[i,k]	d[k,j])	d[i,k]+ d[k,j]	d[i,j]	Min(d[i,j], d[i,k]+ d[k,j])
3	1	1	0	-2	∞	∞	0	0
3	1	2	∞	-2	∞	∞	∞	∞
3	1	3	-2	-2	0	-2	-2	-2
3	1	4	∞	-2	2	0	∞	0
3	2	1	4	2	∞	∞	4	4
3	2	2	0	2	∞	∞	0	0
3	2	3	2	2	0	2	2	2
3	2	4	∞	2	2	4	∞	4
3	3	1	∞	0	∞	∞	∞	∞
3	3	2	∞	0	∞	∞	∞	∞
3	3	3	0	0	0	0	0	0
3	3	4	2	0	2	2	2	2
3	4	1	3	1	∞	∞	3	3
3	4	2	-1	1	∞	∞	-1	-1
3	4	3	1	1	0	1	1	1
3	4	4	0	1	2	3	0	0

K=3

	1	**2**	**3**	**4**
1	0	∞	-2	0
2	4	0	2	4
3	∞	∞	0	2
4	3	-1	1	0

k	i	j	d[i,j]	d[i,k]	d[k,j])	d[i,k]+ d[k,j]	d[i,j]	Min(d[i,j], d[i,k]+ d[k,j])
4	1	1	0	0	3	3	0	0
4	1	2	∞	0	-1	-1	∞	-1
4	1	3	-2	0	1	1	-2	-2
4	1	4	0	0	0	0	0	0
4	2	1	4	4	3	7	4	4
4	2	2	0	4	-1	3	0	0
4	2	3	2	4	1	5	2	2
4	2	4	4	4	0	4	4	4
4	3	1	∞	2	3	5	∞	5
4	3	2	∞	2	-1	1	∞	1
4	3	3	0	2	1	3	0	0
4	3	4	2	2	0	2	2	2
4	4	1	3	0	3	3	3	3
4	4	2	-1	0	-1	-1	-1	-1
4	4	3	1	0	1	1	1	1
4	4	4	0	0	0	0	0	0

K=4

	1	**2**	**3**	**4**
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

Final matrix

	1	**2**	**3**	**4**
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

The cost of moving from 1 to 1 , 2 to 2, ... ie., self loop is 0.

1 to 2 is -1
1 to 3 is -2
1 to 4 is 0
2 to 1 is 4
2 to 3 is 2
2 to 4 is 4

3 to 1 is 5
3 to 2 is 1
3 to 4 is 2
4 to 1 is 3
4 to 2 is -1
4 to 3 is 1