UNIT – IV.

NON LINEAR DATA STRUCTURES – GRAPHS.

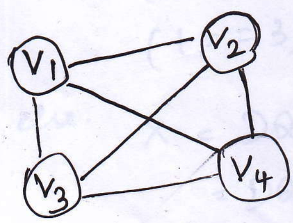
Definition – Representation of graph – Types of graph – Breadth first traversal – Depth-first traversal – Topological sort – BiConnectivity – Cut vertex – Euler circuits – Application of graphs.

DEFINITIONS:

1. GRAPH :-

* A graph $G = (V, E)$ consists of set of vertices V and set of edges E. Vertices are referred to as nodes and the line that connects the nodes are called edges. Each edge is a pair (v, w) where $v, w \in V$.

eg.

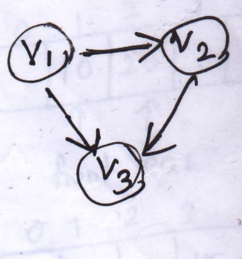


* In the figure, V_1, V_2, V_3 & V_4 are vertices.
* (V_1, V_2), (V_2, V_3), (V_3, V_4), (V_4, V_1), (V_2, V_4), (V_1, V_3) are edges.

2. DIRECTED GRAPH:

* Directed graph is a graph which consists of directed edges, where every edge is unidirectional. If (v, w) is directed edge, then $(v, w) \neq (w, v)$

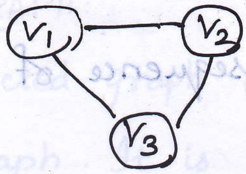
eg.



$(V_1, V_2) \neq (V_2, V_1)$

3. UNDIRECTED GRAPH:

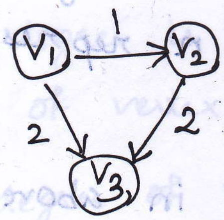
* Undirected graph is a graph which consists of undirected edges. If (v, w) is an undirected edge,

eg. 

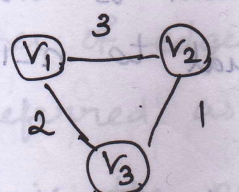
$(V_1, V_2) = (V_2, V_1)$

4. WEIGHTED GRAPH:

* A weighted graph is a graph if every edge in the graph is assigned with a weight or value. It can be either directed or undirected graph.

eg: 

Weighted Directed graph

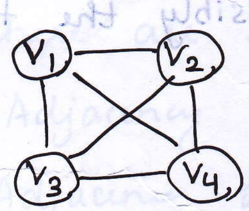


Weighted undirected graph

5. COMPLETE (OR) CONNECTED GRAPH:

* If there is an edge from every vertex to every other vertex in an undirected graph, then it is called complete or connected graph.

* A complete graph with n vertices will have $\frac{n(n-1)}{2}$ no. of edges.

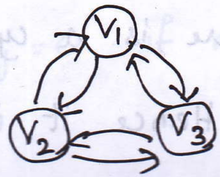
eg: 

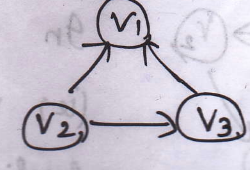
No. of verticesv= 4

No. of edges = $\frac{4 \times 3}{2} = 6$.

6. STRONGLY CONNECTED GRAPH:

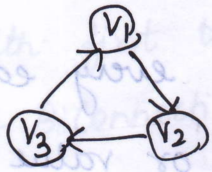
* If there is an edge from every vertex to every other vertex in a directed graph, then it is called strongly connected graph. Otherwise it is weakly connected graph.

eg. 



7. PATH:

* A path in the graph is a sequence of vertices $v_1, v_2, \ldots v_n$ such that $v_i, v_{i+1} \in E$.

eg:  Path from v_1 to v_3 is v_1, v_2, v_3

8. LENGTH OF THE PATH:

* Length of the path is the number of edges on the path, which is equal to $n-1$, where n represents the number of vertices.

* The length of the path v_1 to v_3 in above eg. is 2 (ie) $(v_1, v_2), (v_2, v_3)$.

* If there is a path from vertex to itself, with no edge, then path length is zero.

9. LOOP:

* If graph contains an edge (v, v) from a vertex to itself, then the path is referred to as loop.

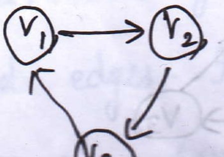
SIMPLE PATH:

* A simple path is a path such that all the vertices on the path are distinct except possibly the first and the last.

10. CYCLE & CYCLIC GRAPH:

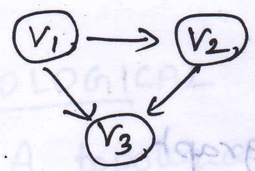
* A cycle is a simple path. A cycle in a graph is a path in which the first and last vertex are same.

* A graph which has cycle is referred to as cyclic graph.

 In this graph, there is a cycle (ie) $v_1 \to v_2 \to v_3 \to v_1$. Hence it is a

11. ACYCLIC GRAPH:

* A directed graph, which has no cycles is referred to as acyclic graph. It is abbreviated as DAG (Directed Acyclic Graph).

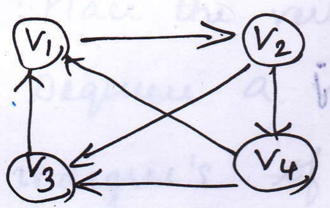


12. DEGREE:

* A degree is the number of edges incident on a vertex. Degree of vertex v is referred as degree(v).

* Indegree of vertex v is the number of edges entering into vertex v.

* Outdegree of vertex v is the number of edges exiting from vertex v.



Indegree $(V_1) = 2$, Outdegree $(V_1) = 1$

Indegree $(V_2) = 1$, Outdegree $(V_2) = 2$

Indegree $(V_3) = 2$, Outdegree $(V_3) = 1$

Indegree $(V_4) = 1$, Outdegree $(V_4) = 2$

REPRESENTATION OF GRAPHS:

* There are two ways to represent a graph.

1. Adjacency Matrix Representation
2. Adjacency List Representation.

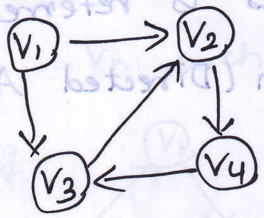
1. Adjacency Matrix Representation.

* Adjacency Matrix A for a graph $G = (V, E)$ with n vertices is a $n \times n$ matrix such that,

$A_{ij} = 1$ if there is an edge from V_i to V_j

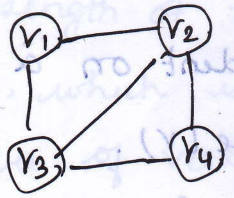
$A_{ij} = 0$ if no edge.

eg1. Adjacency Matrix for directed graph.



	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	0	0	0	1
V_3	0	1	0	0
V_4	0	0	1	0

eg2. Adjacency Matrix for undirected graph



	V_1	V_2	V_3	V_4
V_1	0	1	1	0
V_2	1	0	1	1
V_3	1	1	0	1
V_4	0	1	1	0

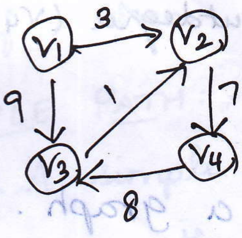
eg3. Adjacency Matrix for Weighted graph

$A_{ij} = C_{ij}$, if there is an edge from V_i to V_j

$C_{ij} \Rightarrow$ weight or cost of edge.

$A_{ij} = 0$, if no edge and $i = j$

$A_{ij} = \infty$, if no edge and $i \neq j$

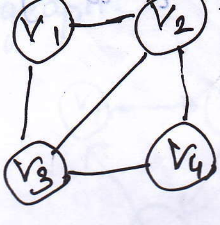


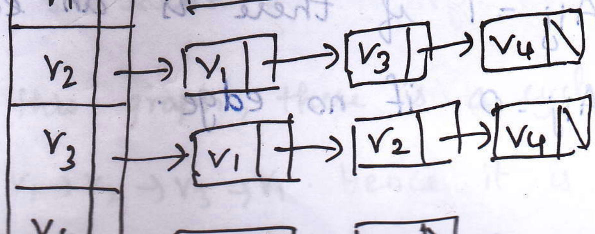
	V_1	V_2	V_3	V_4
V_1	0	3	9	∞
V_2	∞	0	∞	7
V_3	∞	1	0	∞
V_4	∞	∞	8	0

2. Adjacency List Representation:

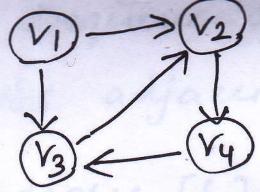
* In this, graph is stored as a linked structure. A list is maintained for all vertices in the graph and then for each vertex, a linked list of its adjacency vertices are maintained.

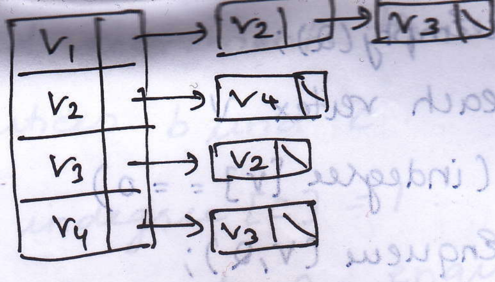
eg1:





eg 2 :





TOPOLOGICAL SORT :

* A topological sort is an 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 ordering.

* Topological ordering is not possible if graph has cycle.

* Steps to perform topological sort :

Step 1 : Find the indegree for each vertex.

Step 2 : Place the vertices whose indegree is 0 on empty queue

Step 3 : Dequeue a vertex v and decrement the indegree's of all its adjacent vertices.

Step 4 : Enqueue the vertex on queue, if its indegree falls to zero.

Step 5 : Repeat from step 3 until queue becomes empty.

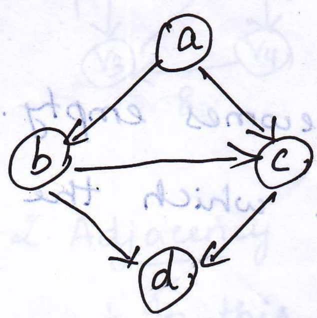
Topological ordering is the order in which the vertices are dequeued.

Algorithm :

```
void TopSort (Graph G)
{
  Queue Q;
  Vertex V,W;
  int count =0;
  Q = CreateQueue (NumofVertex);
```

```
MakeEmpty(Q);
for each vertex V
  if (indegree[v] = = 0)
     Enqueue(v,Q);
  while (!IsEmpty(Q))
  {
     V = Dequeue(Q);
     count++;
      for each W adjacent to V
       if (--indegree[W] = = 0)
          Enqueue(W,Q);
  }
  if (count != NumOfVertex)
     Error("Graph has cycle");
  DisposeQueue(Q);
}
```

Example 1:



Step 1: Find Indegree of all vertices

indegree [a] = 0, indegree [b] = 1, indegree [c] = 2, indegree [d] = 2.

Step 2: Enqueue vertex a whose indegree is 0.

Vertex	Indegree			
a	$\boxed{0}$	0	0	0
b	1	$\boxed{0}$	0	0
c	2	1	$\boxed{0}$	0
d	2	2	1	$\boxed{0}$
Enq:	a	b	c	d

of its adjacent vertices b and c.

indegree [b] = 0 , indegree [c] = 1.

Since indegree of b falls to 0, enqueue b.

Step 4 : Dequeue 'b' and decrement the indegrees of its adjacent vertices c and d.

indegree [c] = 0 , indegree [d] = 1.

Enqueue c.

Step 5 : Dequeue 'c' and decrement indegree of its adjacent vertex 'd'.

indegree [d] = 0.

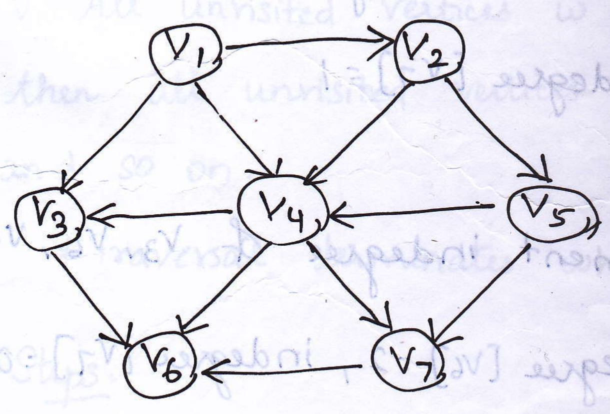
Enqueue d.

Step 6 : Dequeue 'd'. No adjacent vertex for d.

Step 7: As queue is empty, topological ordering gets complete.

Hence, topological ordering for the graph is the order in which vertices are dequeued (ie) a, b, c, d.

Example 2 :



Step 1: Find indegree of all vertices.

indegree [V_1] = 0
indegree [V_2] = 1
indegree [V_3] = 2
indegree [V_4] = 3
indegree [V_5] = 1
indegree [V_6] = 3
indegree [V_7] = 2

Step 2: Enqueue vertex V_1 whose indegree is 0.

Vertices	Indegree						
V_1	$\boxed{0}$	0	0	0	0	0	0
V_2	1	$\boxed{0}$	0	0	0	0	0
V_3	2	1	1	1	$\boxed{0}$	0	0
V_4	3	2	1	$\boxed{0}$	0	0	0
V_5	1	1	$\boxed{0}$	0	0	0	0
V_6	3	3	3	3	2	1	$\boxed{0}$
V_7	2	2	2	1	$\boxed{0}$	0	0
Enq:	V_1	V_2	V_5	V_4	V_3, V_7	–	V_6
Deq:	V_1	V_2	V_5	V_4	V_3	V_7	V_6

Step 3: Dequeue V_1 and decrement indegree of its adjacent vertices V_2, V_3 & V_4.

indegree $[V_2] = 0$, indegree $[V_3] = 1$, indegree $[V_4] = 2$.

∴ Enqueue V_2.

Step 4: Dequeue V_2. Decrement indegree of V_4 & V_5.

indegree $[V_4] = 1$, indegree $[V_5] = 0$.

∴ Enqueue V_5.

Step 5: Dequeue V_5. Decrement indegree of V_4 & V_7

indegree $[V_4] = 0$, indegree $[V_7] = 1$

∴ Enqueue V_4.

Step 6: Dequeue V_4. Decrement indegree of V_3, V_6, V_7

indegree $[V_3] = 0$, indegree $[V_6] = 2$, indegree $[V_7] = 0$.

∴ Enqueue V_3 and V_7.

Step 7: Dequeue V_3. Decrement indegree of V_6.

indegree $[V_6] = 1$.

No vertex to enqueue.

Step 8: Dequeue V_7. Decrement indegree of V_6.

indegree $[V_6] = 0$.

∴ Enqueue V_6.

Step 9: Dequeue V_6. No adjacent vertex to V_6.

Step 10: As queue becomes empty, topological ordering gets completed.

∴ Topological ordering for given graph is

$V_1, V_2, V_5, V_4, V_3, V_7, V_6$

GRAPH TRAVERSAL:

* Graph traversal is a systematic way of visiting vertices in the graph in a specific order.

* There are 2 types of traversal.
  1. Breadth First Search Traversal (BFS).
  2. Depth First Search Traversal (DFS).

BREADTH FIRST SEARCH TRAVERSAL (BFS):

* BFS of a graph G starts from an unvisited vertex V. All unvisited vertices w adjacent to v are visited and then all unvisited vertices w_1 adjacent to w are visited and so on.

* Traversal terminates when no more nodes to visit.

Steps.

1. Choose any node in the graph. Designate it as

2. Find all unvisited adjacent nodes to search node and enqueue them into queue Q. Mark enqueued vertices as visited.

3. Dequeue a node from queue Q. Designate it as new search node.

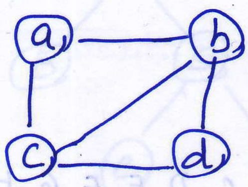
4. Repeat steps 2 and 3 using new search node.

5. Process continues until queue Q becomes empty.

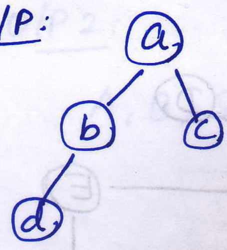
<u>Algorithm</u>:

```
void BFS (Graph G)
{
   Queue Q;
   vertex V, W;
   Q = CreateQueue (NumOfVertex);
  MakeEmpty (Q);
  visited [V] = 1;
  Enqueue (V, Q);
  while (! IsEmpty (Q))
   {
     V = Dequeue [Q);
     print V;
     for all vertices W adjacent to V
      if (visited [W] = = 0) then
       {
         Enqueue (W, Q);
         visited [W] = 1;
       }
   }
}
```

Example I :



O/P:



a, b, c, d

Step 1 :

vertex 'a' is selected as search node :

So visited [a] = 1 and 'a' is enqueued.

a	b	c	d
1	0	0	0

visited

a	

Step 2 : 'a' is dequeued.

'b' & 'c' are adjacent vertices to 'a'.

visited [b] and visited [c] are 0 . So both 'b' & 'c' are enqueued and their visited set to 1.

visited [b] = 1 & visited [c] = 1.

Queue:

b	c

visited

a	b	c	d
1	1	1	0

Step 3 : 'b' is dequeued.

a, c and d are adjacent vertices to 'b'.

But visited [a] and visited [c] are 1. But ~~to~~ vertex d is unvisited.

visited

a	b	c	d
1	1	1	1

∴ visited [d] = 1 and enqueue 'd'.

Step 4 :

c	d

vertex 'c' is dequeued.

Its adjacent vertices a, b and d are visited.

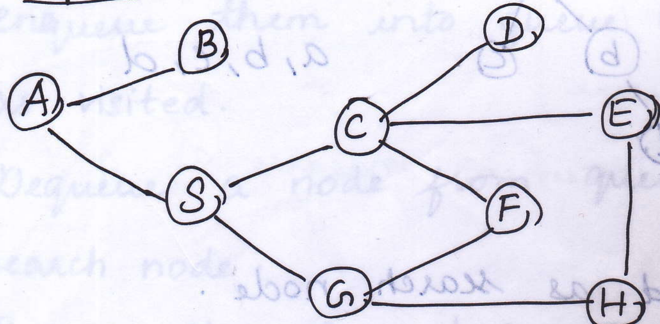
Step 5 :

d

vertex 'd' is dequeued. All its adjacent vertices c & b are visited.

Now, queue becomes empty. The traversal gets

Example 2:



A	B	C	D	E	F	G	H	S
0	0	0	0	0	0	0	0	0

visited

Step 1: vertex 'A' is start vertex.

So visited [A] = 1 and 'A' is enqueued.

visited

A	B	C	D	E	F	G	H	S
1	0	0	0	0	0	0	0	0

Queue: A

Step 2: Dequeue vertex 'A'. 'B' and S are its adjacent unvisited vertices.

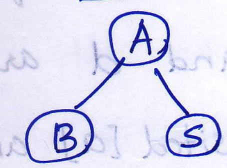
So, visited [B] = 1 and visited [S] = 1 & enqueue B & S.

visited

A	B	C	D	E	F	G	H	S
1	1	0	0	0	0	0	0	1

Queue: B | S

O/P 1:

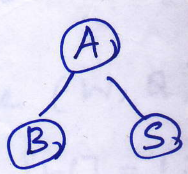


O/P 2: A

Step 3:

Dequeue vertex B. No adjacent vertices to B.

O/P 1:



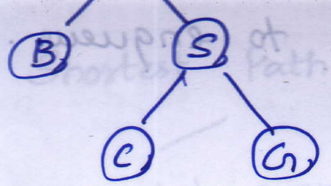
O/P 2: A, B.

Step 4: Dequeue vertex S. C and G are its adjacent unvisited vertices.

So, visited [C] = 1 and visited [G] = 1 & enqueue C, G

visited

A	B	C	D	E	F	G	H	S
1	1	1	0	0	0	1	0	1



A, B, S.

Step 5: Dequeue C. S, D, E and F are C's adjacent vertices where S is already visited and D, E & F are unvisited.

So, visited [D] = 1, visited [E] = 1 and visited [F] = 1 and enqueue D, E, F.

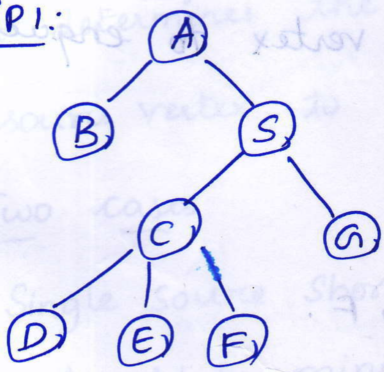
visited

A	B	C	D	E	F	G	H	S
1	1	1	1	1	1	1	0	1

Queue:

G	D	E	F

O/P1:



O/P2:

A, B, S, C

Step 6: Dequeue G. S, F and H are G's adjacent vertices where S and F are visited and H is unvisited.

So, visited [H] = 1 and enqueue H.

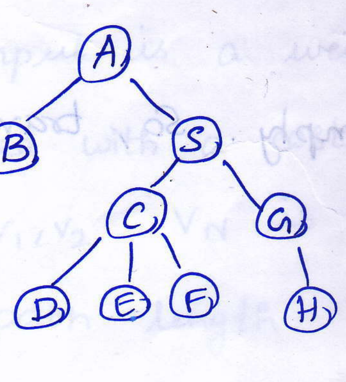
visited

A	B	C	D	E	F	G	H	S
1	1	1	1	1	1	1	1	1

Queue:

D	E	F	H

O/P1:



O/P2:

A, B, S, C, G

Step 7: Dequeue D. C and E are D's adjacent vertices but already visited. So no vertex to enqueue.

Queue: | E | F | H |

O/P1: Same.

O/P2: A, B, S, C, G, D

Step 8: Dequeue E. C and H are E's adjacent vertices but already visited. So no vertex to enqueue.

Queue: | F | H |

O/P1: Same

O/P2: A, B, S, C, G, D, E.

Step 9: Dequeue F. G and C are F's adjacent vertices but already visited. So no vertex to enqueue.

Queue | H |

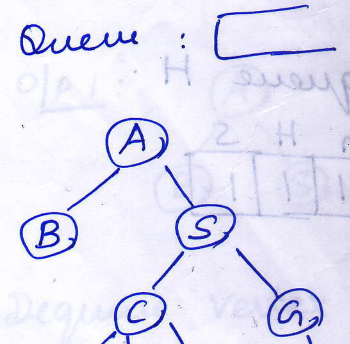
O/P1: Same.

O/P2: A, B, S, C, G, D, E, F.

Step 10: Dequeue H. G and E are H's adjacent vertices but already visited. So, no vertex to enqueue.

Queue: | |

O/P1:



O/P2:

A, B, S, C, G, D, E, F, H.

Step 11: Queue becomes empty. So, traversal gets stopped.

APPLICATIONS OF BFS :

1. Shortest Path Algorithm.
   - UnWeighted graph Alg.
   - Weighted Graph.
     - Single source shortest path Alg. eg. Dijkstra's Alg
     - All-pairs shortest path Alg. eg. Floyd-Warshall Alg.

2. Minimum Spanning Tree.
   - eg 1 : Prim's Alg.
   - eg 2 : Kruskal's Alg

SHORTEST PATH ALGORITHM :

* It determines the minimum cost of the path from the source vertex to every other vertex in graph.

* Two cases

(i) Single source shortest path Algorithm.

It finds minimum cost from single source vertex to all other vertices.

(ii) All-pairs shortest path Algorithm.

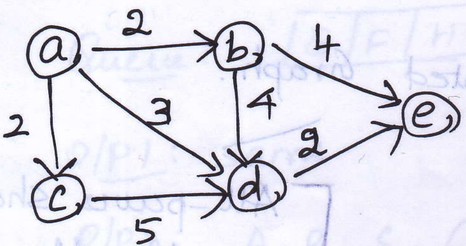
It finds the minimum cost from each vertex to all other vertices.

1. SINGLE SOURCE SHORTEST PATH :

* It works for both weighted and unweighted graph.

* If input is a weighted graph, each edge (v_i, v_j) is associated with a cost $c_{i,j}$ to traverse. The cost of the path $v_1, v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$ referred to as

* Unweighted path length is the number of edges in the path namely N-1.

eg: 

Problem:

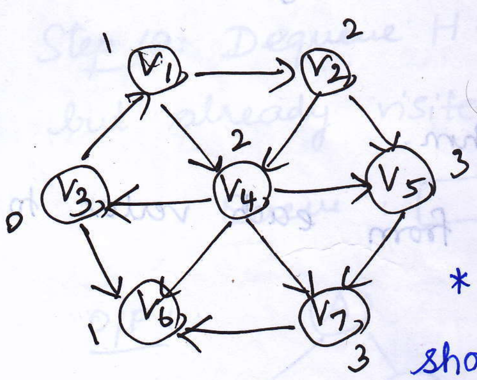
Given as input a weighted or unweighted graph $G = (V, E)$ and a distinguished vertex s as source (or) start vertex, find the shortest path from s to every other vertices in G.

Weighted graph: The shortest weighted path from a to e has a cost of 5 from a to d and d to e.

Unweighted graph: The shortest unweighted path from a to e is 2 from a to b and b to e (or) a to d and d to e.

UNWEIGHTED SHORTEST PATH:

* The figure shows an unweighted graph G.



* using a vertex s as start vertex, find the shortest path from s to all other vertices.

* This is a special case of weighted shortest path since all edges are assigned a weight of 1.

* Suppose V_3 is chosen as start vertex, shortest path from V_3 to V_3 is then a path of length 0.

* Then find all vertices that are at distant 1 away from v_3. They are vertices that are adjacent to v_3. From figure, v_1 and v_6 are one edge from v_3.

* Then find all vertices that are at distant 2 away from v_3. They are vertices adjacent to v_1 and v_6. Thus shortest path to v_2 and v_4 is 2.

* Then find all vertices that are at distant 3 away from v_3. They are vertices adjacent to v_2 and v_4. Thus shortest path to v_5 and v_7 is 3.

* Thus it operates by processing vertices in layers.
  - → vertices are closest to start are evaluated first.
  - → most distant vertices are evaluated last.

Table:

* The algorithm uses a table to find the shortest path.

* The initial configuration of the table.

Vertex.	Known	Dist (d_v)	Path (p_v)
v_1	0	∞	0
v_2	0	∞	0
v_3	0	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

* For each vertex, the table keeps track of 3 pieces of information.

i) dist (d_v) – keeps the distance of that vertex from s. Initially all vertices are unreachable from

(ii) path (p_v) - Entry in P_v is a bookkeeping variable, which is to print the actual path. Entry will be the last vertex to cause a change in d_v of that vertex.

(iii) known - Entry in known is set to 1, after that vertex is processed. Initially all vertices are not known, including start vertex.

When a vertex is marked known, it guarantees that no cheaper path will ever be found and so processing of that vertex is complete.

Algorithm:

Declarations:

```
struct TableEntry
{
  int known;
  DistType dist;
  vertex path;
};
typedef struct TableEntry Table[NumOfVertex];
```

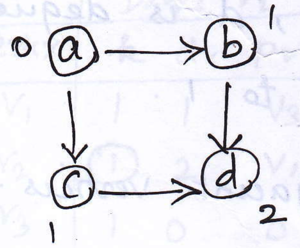
Table Initialization Routine:

```
void InitTable (vertex Start, Graph G, Table T)
{
  int i;
  ReadGraph(G,T);
  for (i=0; i< NumOfVertex ; i++)
  {
    T[i]. known = false ;
    T[i]. dist = Infinity;
    T[i]. path = 0;
```

Alg:

```
void UnWeighted (Table T)
{
  Queue Q;
  Vertex V, W;
  Q = CreateQueue (NumOfVertex);
  MakeEmpty (Q);
  Enqueue (S, Q);
  While (! IsEmpty (Q))
  {
    V = Dequeue (Q);
    T[V]. Known = True;
    for each W adjacent to V
    if (T[W].dist == Infinity)
     {
      T[W].dist = T[V].dist +1;
      T[W]. path = V;
      Enqueue (W, Q);
     }
  }
  DisposeQueue (Q);
}
```

Example 1:



* Start vertex is 'a'. To find the shortest path from vertex a to all other vertices.

Initial configuration,

Vertex	k	dv	pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enq : a

* In next step, vertex a is dequeued.
  Its known is set to 1. Its adjacent vertices are b & c. Its dv is ∞. So update dv.

Deq : a

Vertex	k	dv	pv
a	1	0	0
b	0	1	a
c	0	1	a
d	0	∞	0

Enq: b, c

* In next step, vertex b is dequeued.
  Its known set to 1. Its adjacent vertex is d. Since d's dv is ∞, it is updated.

Deq : b

Vertex	k	dv	pv
a	1	0	0
b	1	1	a
c	0	1	a
d	0	2	b

Enq : c, d

* In next step, vertex c is dequeued. Its known set to 1.
  It's adjacent vertex is d. Since d's dv is not ∞, no change.

Deq : c

Vertex	k	dv	pv
a	1	0	0
b	1	1	a
c	1	1	a
d	0	2	b

* In next step, d is dequeued.
  Its known set to 1.
  It has no adjacent vertices.
  Queue becomes empty. So process gets completed.

Deq : d			
Vertex	k	dv	pv
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b
Enq : -			

Result :

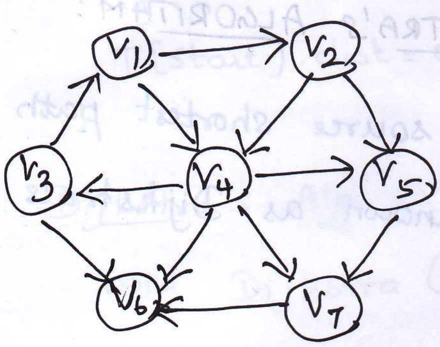
Shortest path from a

a - b is 1 => direct path

a - c is 1 => direct path

a - d is 2 => path is a - b - d

Example 2 :



Start vertex is V_3. To find the shortest path from vertex V_3 to all other vertices.

	Initial State			Deq : V3			Deq : V1			Deq : V6		
Vertex	k	dv	pv	k	dv	pv	k	dv	pv	k	dv	pv
V_1	0	∞	0	0	1	V_3	1	1	V_3	1	1	V_3
V_2	0	∞	0	0	∞	0	0	2	V_1	0	2	V_1
V_3	0	0	0	1	0	0	1	0	0	1	0	0
V_4	0	∞	0	0	∞	0	0	2	V_1	0	2	V_1
V_5	0	∞	0	0	∞	0	0	∞	0	0	∞	0
V_6	0	∞	0	0	1	V_3	0	1	V_3	1	1	V_3
V_7	0	∞	0	0	∞	0	0	∞	0	0	∞	0
	Enq : V_3			Enq : V_1, V_6			Enq : V_6, V_2, V_4			Enq : V_2, V_4		

	Deq : V2			Deq : V4			Deq : V5			Deq : V7		
Vertex	k	dv	pv	k	dv	pv	k	dv	pv	k	dv	pv
V_1	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3
V_2	1	2	V_1	1	2	V_1	1	2	V_1	1	2	V_1
V_3	1	0	0	1	0	0	1	0	0	1	0	0
V_4	0	2	V_1	1	2	V_1	1	2	V_1	1	2	V_1
V_5	0	3	V_2	0	3	V_2	1	3	V_2	1	3	V_2
V_6	1	1	V_3	1	1	V_3	1	1	V_3	1	1	V_3
V_7	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Result: Shortest path from V_3

V_3 to $V_1 \Rightarrow 1 \Rightarrow V_3 - V_1$

V_3 to $V_2 \Rightarrow 2 \Rightarrow$ path $V_3 - V_1 - V_2$

V_3 to $V_4 \Rightarrow 2 \Rightarrow$ path $V_3 - V_1 - V_4$

V_3 to $V_5 \Rightarrow 3 \Rightarrow$ path $V_3 - V_1 - V_2 - V_5$

V_3 to $V_6 \Rightarrow 1 \Rightarrow$ path $V_3 - V_6$

V_3 to $V_7 \Rightarrow 3 \Rightarrow$ path $V_3 - V_1 - V_4 - V_7$

WEIGHTED GRAPH:

SINGLE SOURCE SHORTEST PATH - DIJKSTRA'S ALGORITHM:

* General method to solve single source shortest path problem in a weighted graph is known as Dijkstra's Algorithm.

* It is based on Greedy Algorithm. It solves problem in stages by doing what appears to be the best thing at each stage.

* It Selects a vertex v, which has the smallest d_v among all unknown vertices and declares that the shortest path from s to v is known.

* It updates $d_w = d_v + C_{v,w}$ if new value for d_w would be less value than the old value.

Algorithm:

Declarations:

```
struct TableEntry
{
  int known;
  DistType dist;
  vertex path;
```

typedef struct TableEntry Table[NumOfVertex];

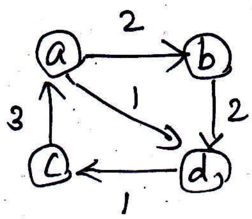
Table Initialization :

```
void InitTable (vertex Start, Graph G, Table T)
{
  int i;
  ReadGraph (G,T);
  for (i=0; i< NumOfVertex; i++)
  {
    T[i].known = 0;
    T[i].dist = infinity;
    T[i].path = 0;
  }
  T[Start].dist = 0;
}
```

Dijkstra's Alg:

```
void Dijkstra (Table T)
{
  vertex V, W;
  for ( ; ; )
  {
    V = Smallest distance unknown vertex;
    T[V].known = True;
    for each W adjacent to V
      if (!T[W].known)
        if (T[V].dist + Cv,w < T[W].dist)
        {
          T[W].dist = T[V].dist + Cv,w;
          T[W].path = V;
        }
  }
}
```

Example 1:



Initial Configuration

Vertex	Known	dv	pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

* a is chosen as source vertex and is marked known.
* b and d are its adjacent vertices and are updated.

Vertex	k	dv	pv
a	1	0	0
b	0	2	a
c	0	∞	0
d	0	1	a

* Select a unknown verstex with smallest distance. Hence d is chosen. set its known to 1. Its adjacent vertex c is updated.

Vertex	k	dv	pv
a	1	0	0
b	0	2	a
c	0	2	d
d	1	1	a

* Next vertex b is chosen. Its known set to 1. Its adjacent vertex is d but already visited.

vertex	k	dv	pv
a	1	0	0
b	1	2	a
c	0	2	d
d	1	1	a

* Next vertex c is chosen. Its known is set to 1. It adjacent is a already visited.

Vertex	k	dv	pv
a	1	0	0
b	1	2	a
c	1	2	d
d	1	1	a

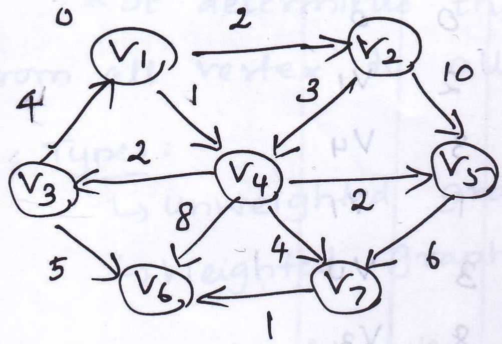
Result:

a – b ⇒ 2 ⇒ direct path

a – c ⇒ 2 ⇒ a → d → c

a – d ⇒ 1 ⇒ direct path.

Example 2:



V_1 is taken as source vertex and is marked known. Set its known to 1 and update distance & path of its adjacent.

Initial Configuration:

Vertex	k	dv	Pv
V_1	0	0	0
V_2	0	∞	0
V_3	0	∞	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

vertex	k	dv	Pv
V_1	1	0	0
V_2	0	2	V_1
V_3	0	∞	0
V_4	0	1	V_1
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

* Next unknown smallest distance vertex V_4 is chosen. Its known set to 1 and its adjacent vertices are updated.

Vertex	k	dv	pv
V_1	1	0	0
V_2	0	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

* Next vertex V_2 is chosen. V_4 & V_5 are adjacent of V_2. Their distances are not update b'coz new values are greater than old values.

vertex	k	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

Vertex	k	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	8	v_3
v_7	0	5	v_4

Vertex	k	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	8	v_3
v_7	0	5	v_4

Vertex	k	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	6	v_7
v_7	1	5	v_4

Vertex	k	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	1	6	v_7
v_7	1	5	v_4

Shortest distance from v_1:

$v_1 - v_2$ is 2 $\Rightarrow$ direct path $v_1 \to v_2$

$v_1 - v_3$ is 3 $\Rightarrow$ $v_1 \to v_4 \to v_3$

$v_1 - v_4$ is 1 $\Rightarrow$ direct path $v_1 \to v_4$

$v_1 - v_5$ is 3 $\Rightarrow$ $v_1 \to v_4 \to v_5$

$v_1 - v_6$ is 6 $\Rightarrow$ $v_1 \to v_4 \to v_7 \to v_6$

$v_1 - v_7$ is 5 $\Rightarrow$ $v_1 \to v_4 \to v_7$