UNIT:4

GRAPHS

Graph Definition:

A Graph 'G' = (V, E) consists of a set of Vertices, V, and a Set of edges, E.

Each edge is a pair (v, w) where, $v, w \in V$. Edges are Sometimes referred as Arcs.

Terminologies:

a) If the pair (v, w) is ordered, then the graph is directed. Directed graphs are sometimes referred as digraphs. Vertex 'w' is adjacent to v if and only if $(v, w) \in E$.

b) In an Undirected graph with edge (v, w), and hence (w, v), w is adjacent to v and v is adjacent to w. Sometimes an Edge has a third Component, known as either a weight or a Cost.

c) A path in a graph is a Sequence of Vertices $w_1, w_2, w_3, \ldots w_N$. Such that $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N$. The length of Such a path is the number of edges on the path, which is equal to $N-1$.

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

e) A cycle is a path in a digraph that has at least three vertices such that the last vertex on the path is adjacent to

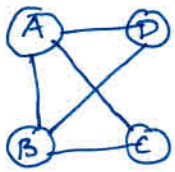
f) A directed graph is acyclic if it has no cycles. A directed acyclic graph is abbreviated as DAG.

g) An undirected graph is connected if there is path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

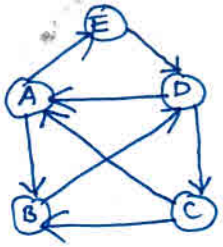
h) If a directed graph is not strongly connected, but the graph is connected, then the graph is said to be weakly connected.

i) A complete graph is a graph in which there is an edge between every pair of vertices.

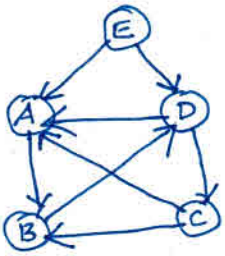
Examples: Connected Graphs



(connected)



(Strongly Connected)



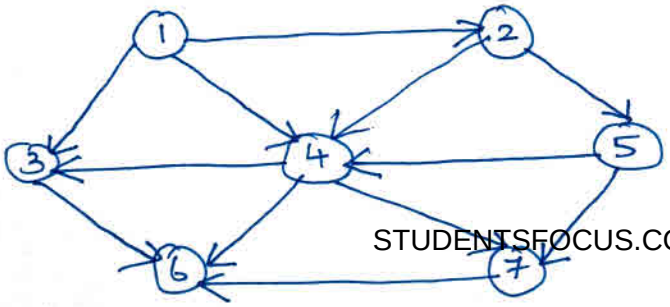
(Weakly Connected)

Representation of Graphs:

We will consider directed graphs.

Example:

A Directed Graph



This graph represents 7 vertices and 12 edges.

a) one simple way to represent a graph is to use a two dimensional array. This is known as an adjacency matrix representation.

b) For each edge (U,V), we set $A[U][V] = 1$, otherwise the entry in the array is 0. If the edge has a weight associated with it, then we can set $A[U][V]$ equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.

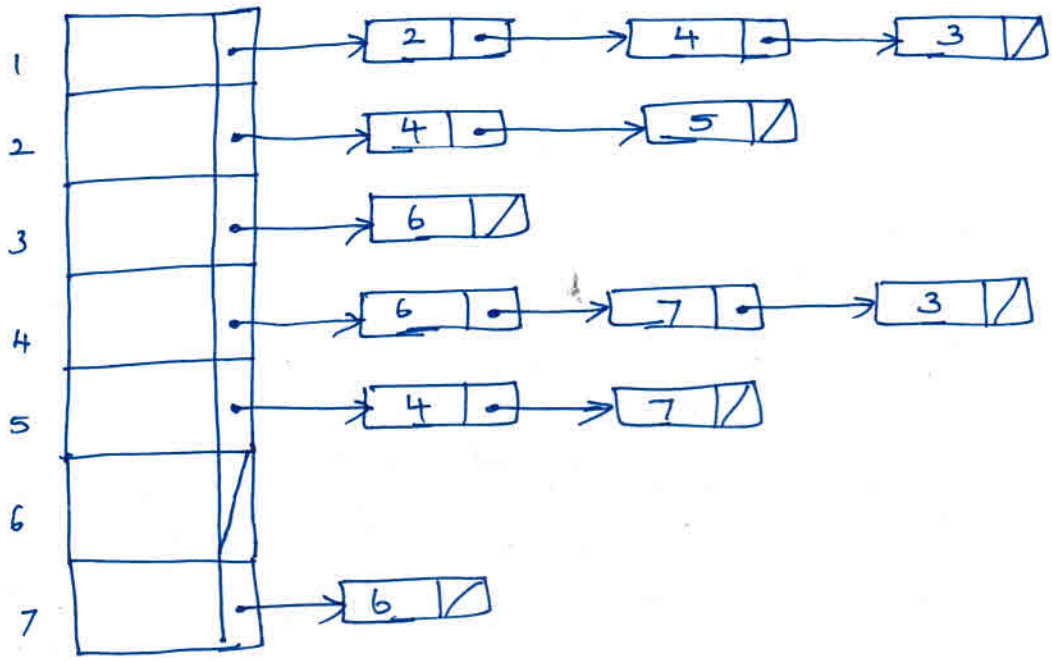
c) Even though the array implementation is simple, the space requirement is square of the number of vertices. This can be prohibitive if the graph does not have many edges. Adjacency matrix is an appropriate representation if the graph is dense (i.e. $|E| = |V|^2$)

d) If the graph is not dense, in other words, if the graph is sparse, a better solution is an adjacency list representation.

e) For each vertex, we keep a list of all adjacent vertices. The space requirement is then $O(|E| + |V|)$.

f) Adjacency lists are the standard way to represent graphs. Undirected graphs can be similarly represented. each edge (U,V) appears in two lists, so the space usage essentially doubles.

An Adjacency list representation of a graph



Topological Sort

a) A topological sort is an ordering of vertices in a Directed Acyclic Graph (DAG), such that if there is a Path from V_i to V_j, then V_j appears after V_i in the ordering.

b) Topological Sorting 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.

c) Furthermore, the ordering is not Unique.

d) A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges. We can then Print this vertex, and remove it, along with its edges, from the graph. Then we apply this same Strategy to the rest of the graph.

e) To formalize this, we define the 'indegree' of a vertex 'v' as the number of edges (u,v). We compute the indegrees of all vertices in the graph.

f) Assuming that the indegree array is initialized and that the graph is read into an adjacency list, we can then apply the following algorithm to generate a topological ordering.

Simple topological sort pseudocode

```
Void Topsort (Graph G)
{
    int Counter;
    Vertex V, W;
    for (Counter = 0; Counter < numvertex; Counter++)
    {
        V = FindNewVertexOfIndegreeZero( );
        if ( V == NotAVertex)
        {
            Error (" Graph has a cycle");
            break;
        }
        TopNum[V] = Counter;
        for each W adjacent to V
            Indegree [W]--;
    }
}
```

g) The Function FindNewVertexOfIndegreeZero scans the indegree array looking for a vertex with indegree 0 that has not already been assigned a topological number. It returns NOTAVertex if no such vertex exists, this indicates that the graph has a cycle.

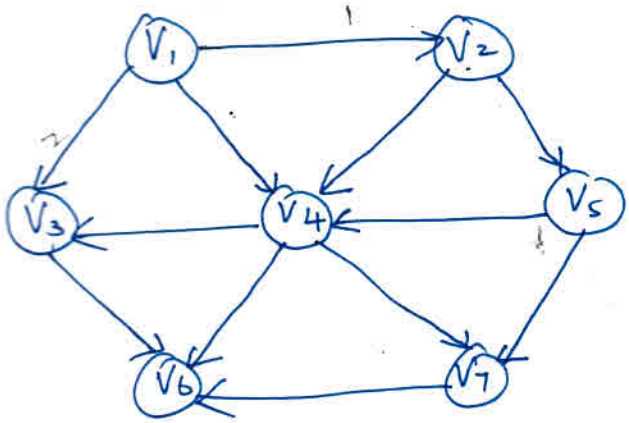
h) Because "FindNewVertexOfIndegreeZero" is a simple sequential scan of the Indegree array, each call to it takes O(|V|) time. Since there are |V| such calls, the running time of the algorithm is $O(|V|^2)$.

i) If the graph is sparse, we would expect that only a few vertices have their indegrees updated during each iteration. However, in search of vertex with indegree 0, we look at all vertices, eventhough only a few have changed.

j) We can remove this inefficiency by keeping all the (unassigned) vertices of indegree 0 in a special box. The 'FindNewVertexOfIndegreeZero' function then returns (and removes) any vertex in the box. We then decrement the indegrees of the adjacent vertices, we check each vertex and place it in the box if its indegree falls to 0.

k) To implement the box, we can use either a stack or queue. First, the indegree is computed for every vertex. Then all vertices of indegree 0 are placed on an initially empty queue. While the queue is not empty, a vertex v' is removed and all edges adjacent to v' have their indegrees

Example:- Consider the Acyclic Graph



Result of applying Topological Sort to the above graph:

	Indegree Before Dequeue						
vertex	1	2	3	4	5	6	7
V_1	0	0	0	0	0	0	0
V_2	1	0	0	0	0	0	0
V_3	2	1	1	1	0	0	0
V_4	3	2	1	0	0	0	0
V_5	-1	1	0	0	0	0	0
V_6	3	3	3	3	2	1	0
V_7	2	2	2	1	0	0	0
Enqueue	V_1	V_2	V_5	V_4	V_3, V_7		V_6
Dequeue	V_1	V_2	V_5	V_4	V_3	V_7	V_6

Routine to Perform Topological Sort

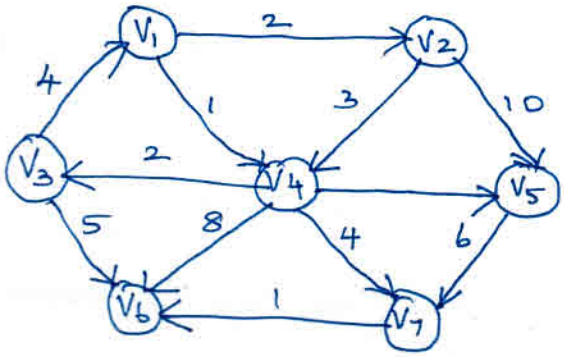
```
Void Topsort( Graph G)
{
    Queue Q;
    int Counter = 0;
    Vertex V, W;
    Q = CreateQueue(NumVertex);
    makeEmpty(Queue);
    For each vertex V
      if (Indegree[V] == 0)
          Enqueue(V, Q);
    While (!isEmpty(Q))
    {
      V = Dequeue(Q);
      TopNum[V] = ++Counter;
      for each W adjacent to V
        if (--Indegree[W] == 0)
            Enqueue(W, Q);
    }
    if (Counter != NumVertex)
      Error("Graph has a cycle");
    DisposeQueue(Q);
}
```

Shortest Path Algorithms

Single Source shortest path Problem

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

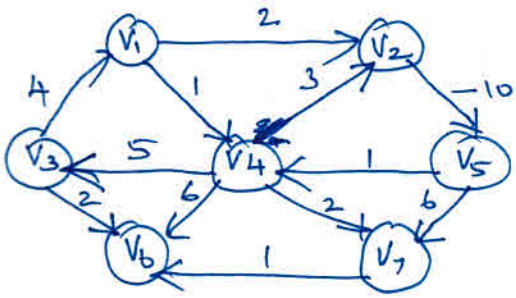
Example:1



A directed graph G.

a) In the above graph, the shortest weighted path from V_1 to V_6 has a cost of 6 and goes from V_1 to V_4 to V_7 to V_6. The shortest unweighted path between V_1 and V_6 is 2. (ie. Number of Edges from V_1 to V_6).

Example: 2



a) The graph in the previous Example:1 has no edges of negative cost. But the graph in Example:2 has edge with negative cost.

b) The path from V_5 to V_4 has cost 1, but a shorter path exists

c) This path is still not shortest, because we could stay in the loop arbitrarily long. Thus, the shortest ~~point~~ path between there two points is Undefined. This loop is known as a negative cost cycle, when present in the graph, the shortest Path are not defined.

d) Four Versions of shortest path algorithms we will explore,

i) Algorithm for Unweighted shortest path

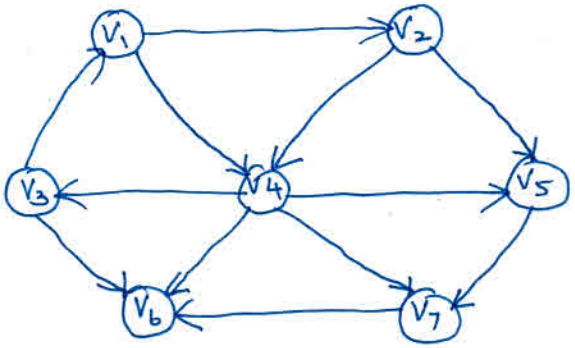
ii) Weighted Shortest path Algorithm (Dijkstra's Algorithm) with no Negative Cost edges

iii) Algorithm for Graph with Negative Cost Edges

iv) Algorith for shortest path in Acyclic Graph.

Algorithm for Unweighted Shortest paths:

Example:



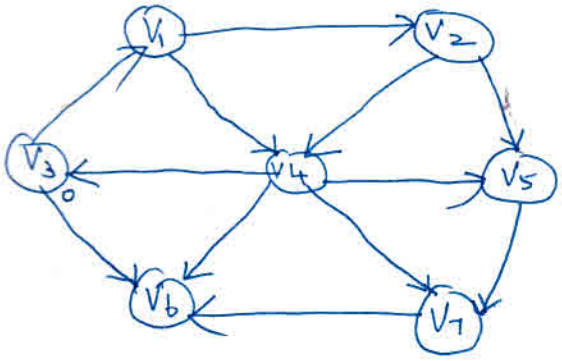
An Unweighted directed Graph G

a) Using Some Vertex 's', which is an input Parameter, we would like to find Shortest path from 's' to all other Vertices.

b) we are only interested in the number of edges contained on the Path, so there are no weights on the edges. This is clearly a Special case of the weighted shortest path Problem, Since we could assign all edges a weight of 1.

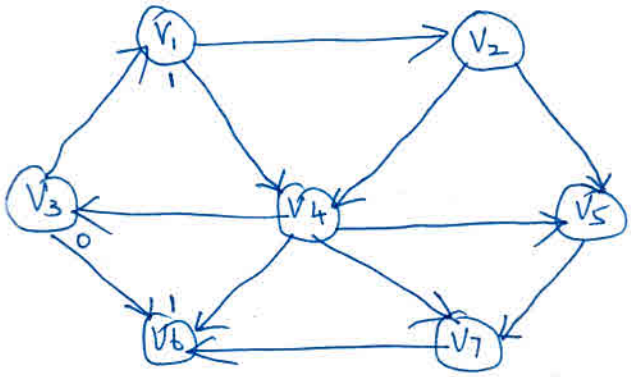
c) Suppose we ~~should~~ choose 's' to be V_3. Immediately, we can tell that the Shortest path from s to V_3 is then a path length of 0.

Graph after marking the Start node as reachable in zero edges



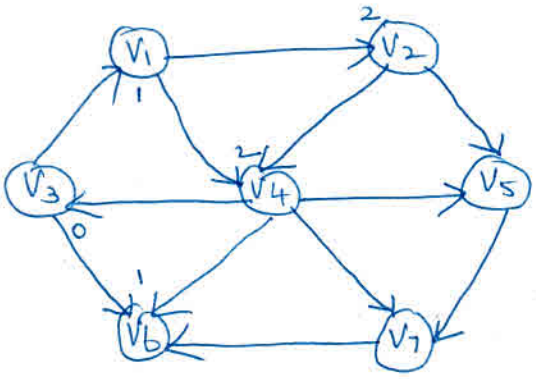
d) Now we can start looking for all vertices that are at distance 1 away from s. This can be found by looking at the vertices that are adjacent to s. If we do this, we see that V_1 and V_6 are one edge from s.

Graph after finding all vertices whose path length from s is 1



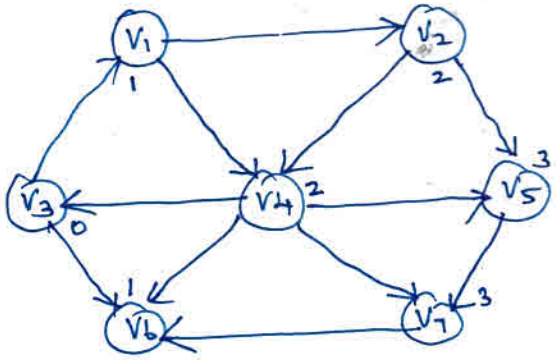
e) We can now find vertices whose shortest path from s is exactly 2, by finding all the vertices adjacent to V_1 and V_6, whose shortest paths are not already known. This search tells us that the shortest path to V_2 and V_4 is 2.

Graph after finding all vertices whose shortest path is 2



f) Finally, we can find, by examining vertices adjacent to the recently evaluated V_2 and V_4, that V_5 and V_7 have a shortest path of three edges

Final shortest paths



g) This strategy of searching a graph is known as Breadth-First search. It operates by processing vertices in layers, the vertices closest to the start are evaluated first, and the most distant vertices are evaluated last. This is similar to the level order traversals of trees.

h) We will maintain an Initial Configuation of a table for our algorithm, which keep track of three information for each vertex

i) First, we will keep the vertex distance from s in the entry dv.

iii) The entry known is set to 1 after a vertex is processed.

Note:

1. Initially all vertices are unreachable except for s, whose path length is 0.
2. Initially, all entries are not known, including the start vertex.
3. When a vertex is marked known, we have a guarantee that no cheaper path will ever be found, and so processing for that vertex is essentially complete.

Initial Configuration of table used in Un-weighted Shortest Path Computation.

V	known	d_v	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

Sequence of Data changes during the Unweighted Shortest-Path Algorithm.

V	Initial state			V_3 Dequeued			V_1 Dequeued			V_6 Dequeued		
	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v	known	d_v	P_v
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_7	0	∞	0	0	1	V_3	0	1	V_3	1	1	V_3
								∞	0	0	∞	0

Table Continued

V	V_2 Dequeued			V_4 Dequeued			V_5 Dequeued			V_7 dequeued		
	known	dv	Pv	known	dv	Pv	known	dv	Pv	known	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	0	∞	0	0	3	V_4	0	3	V_4	1	3	V_4
Q	V_4, V_5			V_5, V_7			V_7			Empty		

Routine for Unweighted Shortest Path Algorithm.

```
Void unweighted (Table T)
{
    Queue Q;
    Vertex V, W;
    Q = CreateQueue(NumVertex);
    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);
          }
    }
```

Dijkstra's Algorithm

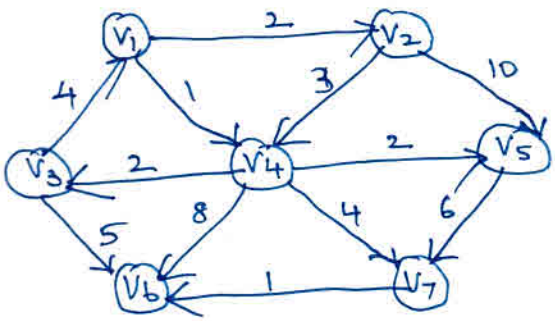
a) If the graph is weighted, the problem becomes harder, but can still use the ideas from the Unweighted case.

b) We keep all of the same information as before. Thus, each vertex is marked as either known or unknown. A tentative distance d_v is kept for each vertex, as before. As before, we record p_v, which is the last vertex to cause a change to d_v.

c) The general method to solve the Single-Source shortest path Problem is known as Dijkstra's algorithm. This algorithm works on the principle of Greedy algorithm (i.e. Solving a problem in stages by doing what appears to be the best thing at each stage.)

d) Dijkstra's algorithm proceeds in stages, just like the unweighted Shortest-Path algorithm. At each stage, Dijkstra's algorithm selects a vertex 'v' which has the smallest d_v among all the unknown vertices, and declares that the shortest path from s to v is known.

Example:



Initial configuration of table used in Dijkstra's Algorithm.

V	known	d_v	P_v
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

Table Represents the initial Configuration, assuming that the start node, s, is V_1.

After V_1 is declared known

V	known	d_v	P_v
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

After V_4 is declared known

V	known	d_v	P_v
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

After V_2 is declared known

V	known	d_v	P_v
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

After V_3 is declared known

V	known	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

After V_5 declared known

V	known	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

After V_7 declared known

V	known	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

Finally After V_6 declared known

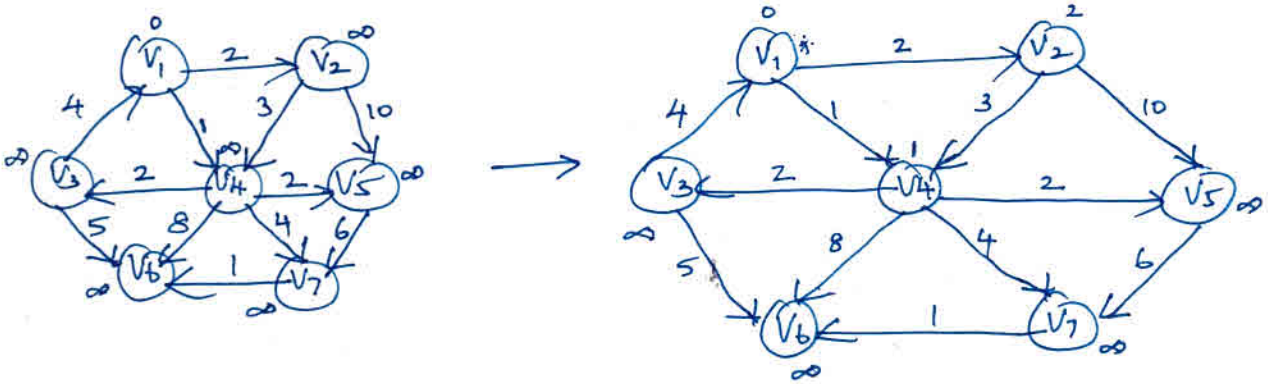
V	known	d_v	P_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	1	6	V_7
V_7	1	5	V_4

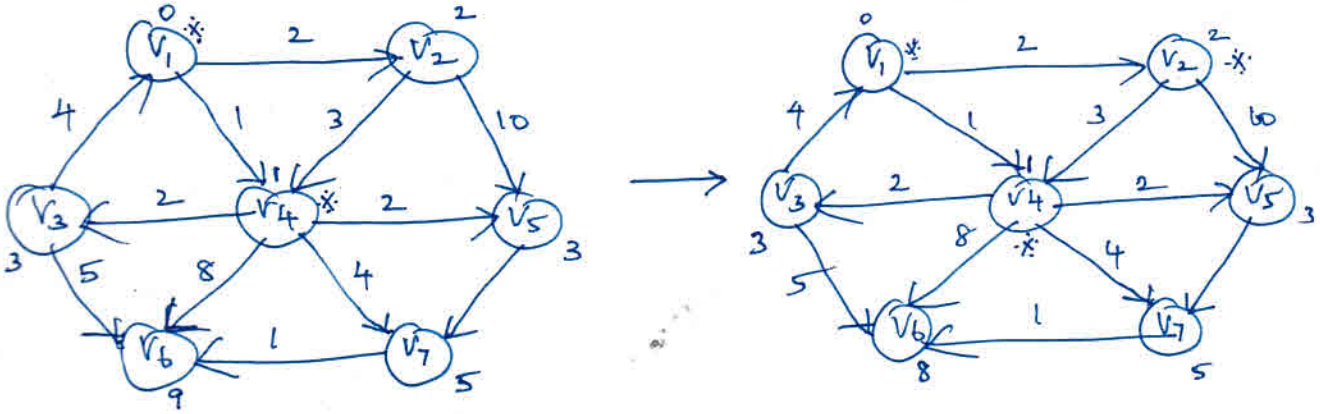
Algorith terminates.

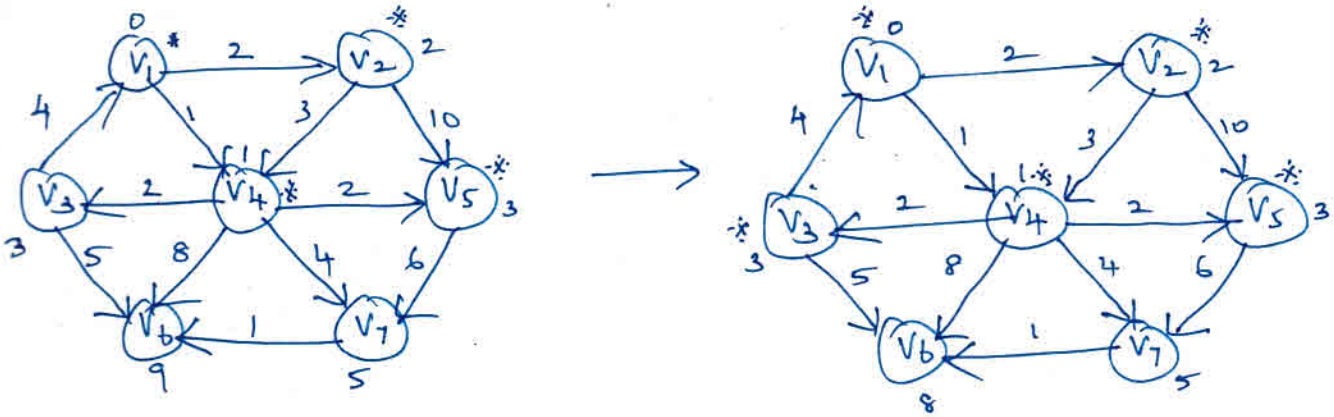
Routine for Dijkstra's Algorithm

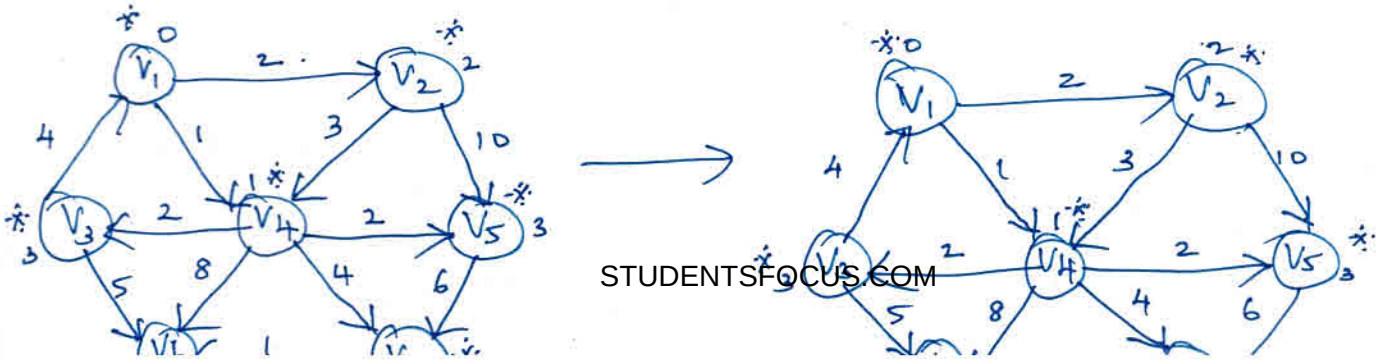
```
Void Dijkstra (Table T)
{
    Vertex V, W;
    for (; ;)
    {
        V = Smallest Unknown distance vertex;
        if (V == NotAVertex)
            break;
        T[V].known = True;
        for each W adjacent to V
            if( !T[W].known)
                if(T[V].Dist + CVW < T[W].Dist)
                {
```

Stages of Dijkstra's Algorithm:









Graphs with Negative Edge costs

a) If the graph has negative edge costs, then Dijkstra's algorithm does not work.

b) The problem is that once a vertex 'u' is declared known, it is possible that from some other, unknown vertex 'v' there is a path back to 'u' that is very negative. In such a case, taking a path from s to v back to u is better than going from s to u without using v.

c) A tempting solution is to add a constant Δ to each cost, thus removing negative edges, calculate a shortest path on the new graph, and then use the result on the original. This strategy does not work because paths with many edges become more weighty than paths with new edges.

d) A combination of the weighted and unweighted algorithms will solve the problem, but at the cost of a drastic increase in running time. we forget about the concept of known vertices, since our algorithm needs to be able to change its mind. we begin by placing 's' on a queue, Then, at each stage, we dequeue a vertex v, we find all vertices w adjacent to v such that $d_w > d_v + c_{v,w}$.

we update d_w and p_w, and place w on a queue if it is not already there. A bit can be set for each vertex to indicate presence in the queue. we repeat this process until the queue is empty.

Routine for weighted Shortest path algorithm with Negative Edge cost

```
Void WeightedNegative (Table T)
{
    Queue Q;
    Vertex V, W;
    Q = CreateQueue (NumVertex);
    makeEmpty (Q);
    Enqueue (S, Q);
    while (!IsEmpty (Q))
    {
        V = Dequeue (Q);
        for each W adjacent to V
        if (T[V].Dist + CVW < T[W].Dist)
        {
            T[W].Dist = T[V].Dist + CVW;
            T[W].path = V;
            if (W is not already in Q)
                Enqueue (W, Q);
        }
    }
    DisposeQueue (Q);
}
```

Acyclic Graphs

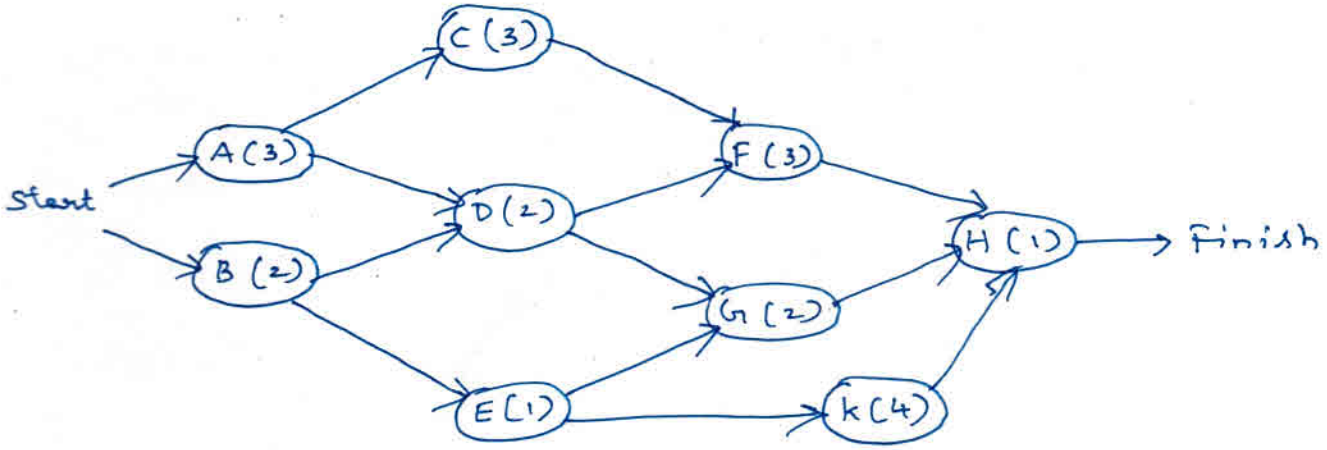
a) If the graph is known to be acyclic, we can improve Dijkstra's algorithm by changing the order in which vertices are declared known, otherwise known as the vertex selection rule.

b) The new rule is to select vertices in topological order.

c) A more important use of acyclic graphs is Critical path Analysis. Each node represents an activity that must be performed, along with the time it takes to complete the activity. This graph is known as Activity-Node Graph. The edges represents precedence relationships. An Edge (v, w) means that activity v must be completed before activity w may begin.

Example:

Activity Node Graph

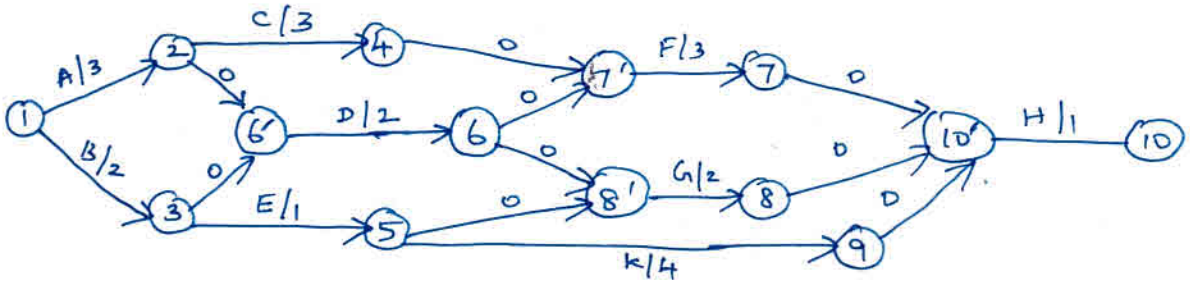


d) To compute Earliest completion time, Latest Completion time and Slack with respect to each activity, we convert the activity node graph to an Event-Node Graph.

e) Each event corresponds to the completion of an activity and all its dependent activities. Events reachable from a

f) Dummy nodes and edges may need to be inserted in the case where an activity depends on several others.

Event Node Graph for our Example



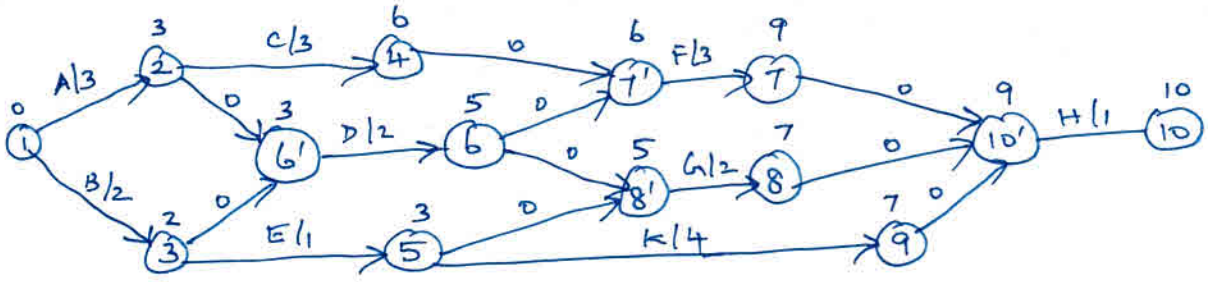
g) To Find the Earliest Completion Time of the graph, we need to find the length of the longest path from the first Event to the last event.

If EC_i is the Earliest Completion time for node, i. then the applicable rules are,

$$EC_1 = 0$$

$$EC_w = \max_{(v,w) \in E} (EC_v + C_{v,w})$$

Graph with Earliest Completion time for our example

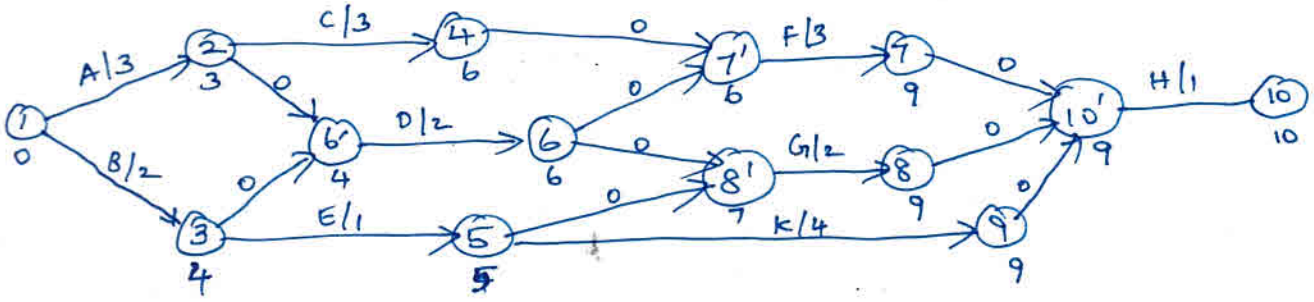


h) We can also Compute latest time, LC_i, that each event can finish without affecting the final Completion time. The formulas are,

$$LC_n = EC_n$$

$$LC_v = \min (LC_w - C_{vw})$$

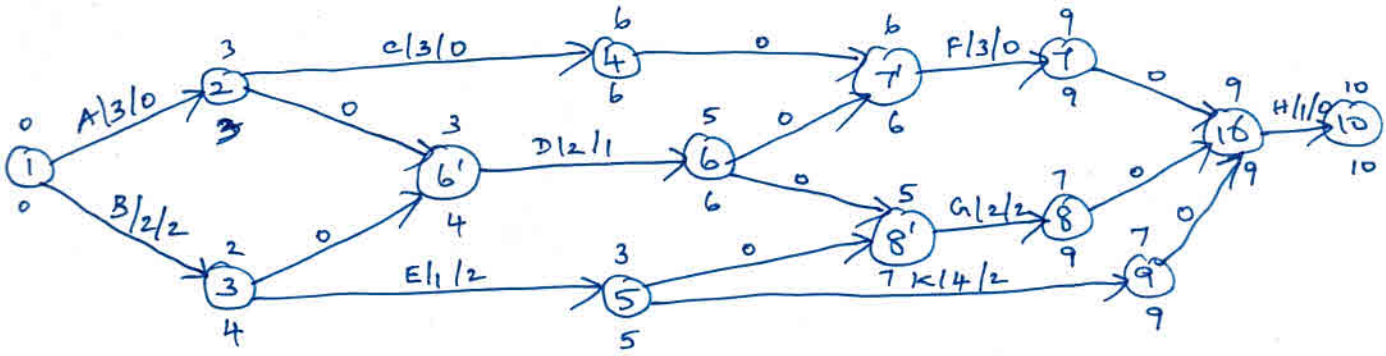
Graph with Latest Completion Time



i) The slack time for each edge in the event node graph represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion.

$$\text{Slack}(v,w) = LC_w - EC_v - C_{vw}$$

Graph with Earliest Completion time, Latest Completion time and slack



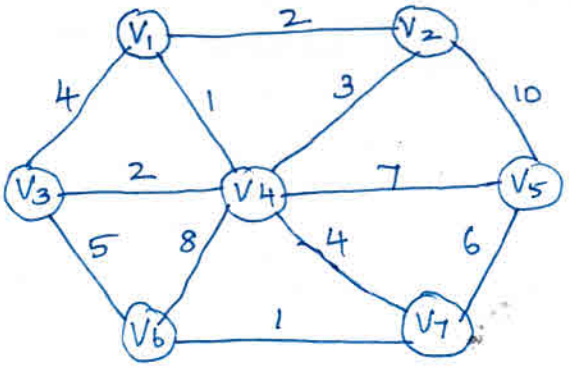
J) Some activities have zero slack. These are Critical activities, which must finish on schedule. There is atleast one Path containing entirely of zero-slack edges, such a Path is a Critical Path.

Minimum Spanning Tree:

a) A minimum spanning tree of an undirected graph 'G' is a tree formed from graph edges that connects all the vertices of 'G' at lowest total cost.

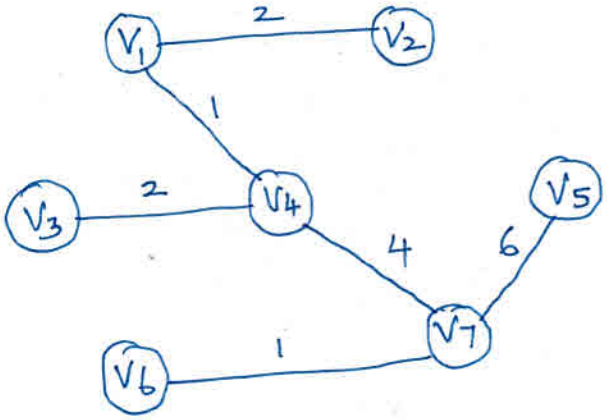
b) A minimum spanning tree exists if and only if 'G' is connected.

Example:



A Graph 'G'

Its minimum spanning tree.



c) Number of Edges in the minimum spanning tree is $|V|-1$.

d) The minimum spanning tree is a tree because it is acyclic, it is spanning because it covers all the vertex, and it is minimum because it covers all the vertex

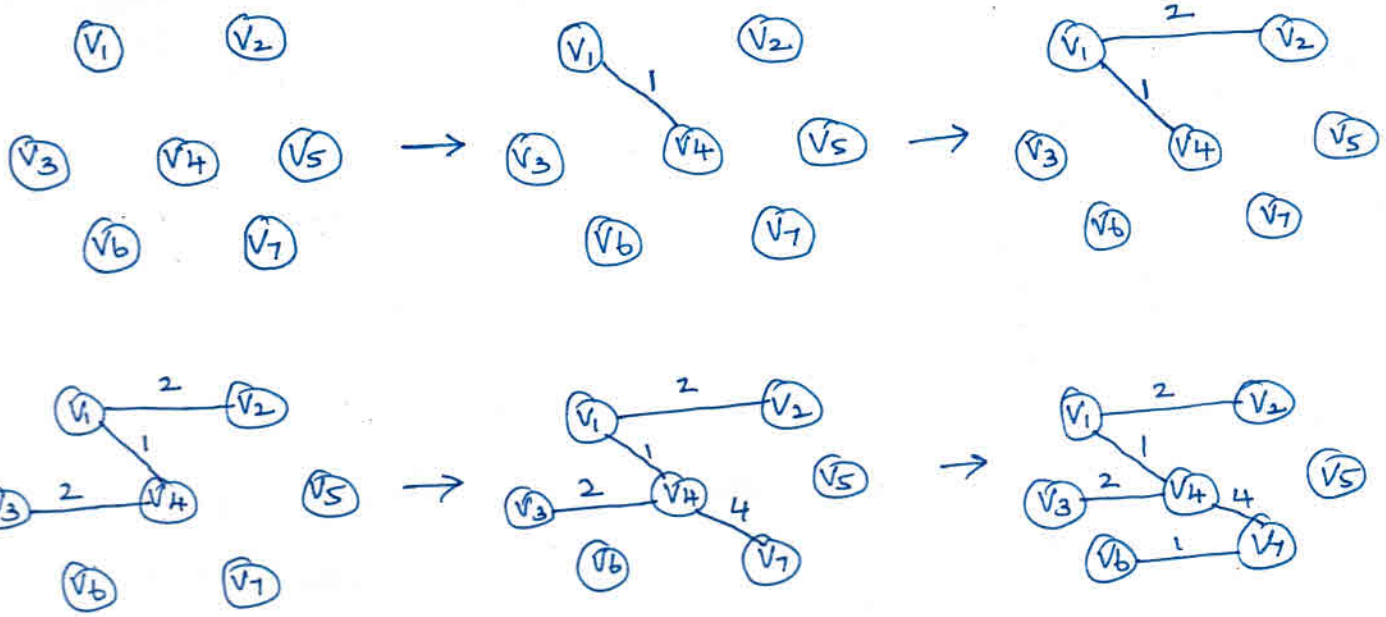
Prim's Algorithm.

a) one way to compute a minimum spanning tree is to grow the tree in successive stages. In each stage, one node is picked as the root, and we add an edge, and thus an associated vertex, to the tree.

b) At any point in the algorithm, we can see that we have a set of vertices that have already been included in the tree, the rest of the vertices have not.

c) The algorithm then finds, at each stage, a new vertex to add to the tree by choosing the edge (u, v) such that the cost of (u, v) is the smallest among all edges where 'u' is in the tree and 'v' is not.

Prim's algorithm after each stage



d) We can see that Prim's algorithm is essentially identical to Dijkstra's algorithm for shortest paths.

e) As before, ~~we~~ for each vertex we keep values d_v, P_v and an indication of whether it is known or unknown.

f) d_v is the weight of the shortest arc connecting v to a unknown vertex and P_v is the last vertex to cause a change in d_v.

g) The update rule is even simpler, after a vertex v is selected, for each unknown 'w' adjacent to v, $d_w = \min(d_w, c_{w,v})$.

Initial configuration of table in Prim's Algorithm

V	known	d_v	P_v
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

After v_1 declared known

V	known	d_v	P_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	4	v_1
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

After v_4 declared known

V	known	d_v	P_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	8	v_4
v_7	0	4	v_4

After v_2 and then v_3 declared known

V	known	d_v	P_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	5	v_3
v_7	0	4	v_4

After v_7 declared known

V	known	d_v	P_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	6	v_7
v_6	0	1	v_7
v_7	1	4	v_4

After v_6 & v_5 declared known

V	known	d_v	P_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	6	v_7
v_6	1	1	v_7
v_7	1	4	v_4

h) The edges in the spanning tree can be read from the table: (v_2, v_1), (v_3, v_4), (v_4, v_1), (v_5, v_7), (v_6, v_7), (v_7, v_4).

Kruskal's Algorithm.

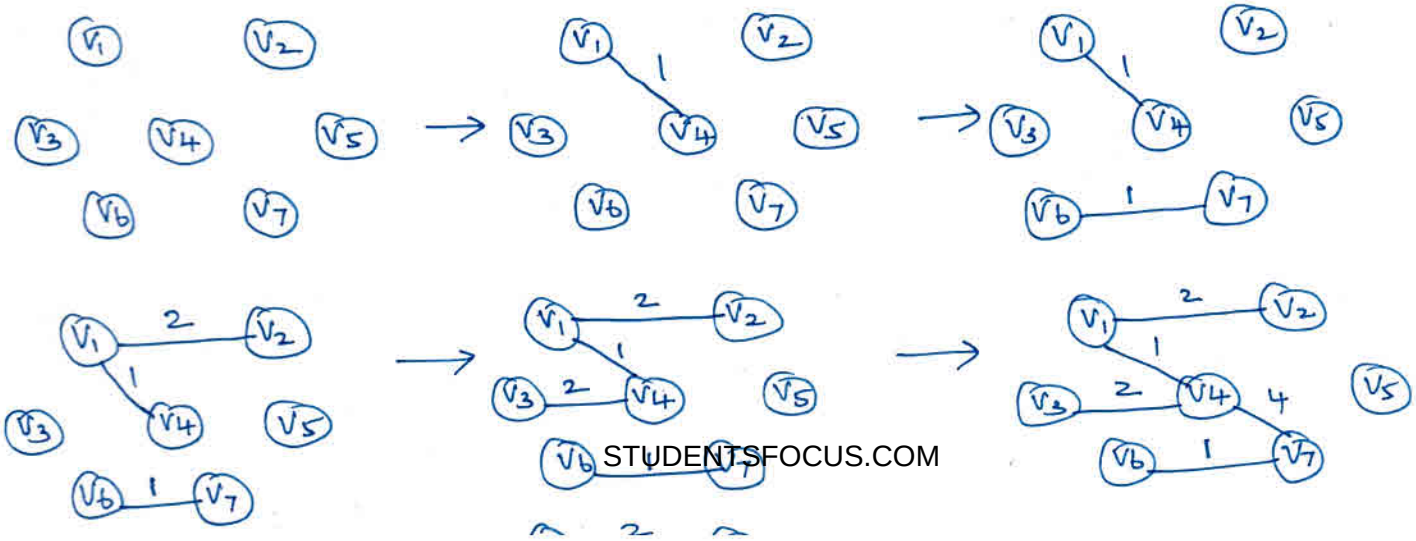
a) A second algorithm for minimum spanning tree is to select the edges in order of smallest weight and accept an edge if it does not cause a cycle.

b) This algorithm maintains a forest (i.e. collection of trees). Initially, there are $|V|$ single-node trees. Adding an edge merges two trees into one. When the algorithm terminates, there is only one tree, and this is minimum spanning tree.

c) The algorithm terminates when enough edges are accepted. It turns out to be simple to decide whether edge (u,v) should be accepted or rejected.

d) At any point in the process, two vertices belong to the same set if and only if they are connected in the current spanning forest. Thus, each vertex is initially in its own set. If u and v are in the same set, the edge is rejected, because they are already connected. otherwise, the edge is accepted, and a union is performed on the two sets containing u and v.

Stages of kruskal's algorithm on Graph G.



Action of Kruskal's Algorithm on G

Edge	weight	Action
(v_1, v_4)	1	Accepted
(v_6, v_7)	1	Accepted
(v_1, v_2)	2	Accepted
(v_3, v_4)	2	Accepted
(v_2, v_4)	3	Rejected
(v_1, v_3)	4	Rejected
(v_4, v_7)	4	Accepted
(v_3, v_6)	5	Rejected
(v_5, v_7)	6	Accepted

Routine for kruskal's Algorithm.

```
Void kruskal (Graph G)
{
   int EdgesAccepted;
   DisjSet S;
   PriorityQueue H;
   Vertex U, V;
   SetType Uset, Vset;
   Edge E;
   Initialize (S);
   ReadGraphIntoHeapArray (G, H);
   BuildHeap (H);
   EdgesAccepted = 0;
   while (EdgesAccepted < NumVertex - 1)
   {
      E = DeleteMin(H);
      Uset = Find (U, S);
      Vset = Find (V, S);
      if (Uset != Vset)
      {
         EdgesAccepted++;
         SetUnion (S, Uset, Vset);
      }
   }
}
```

Depth First Traversal

a) Depth first traversal (search) is a generalization of Pre-order traversal. Starting at some vertex, V, we process 'V', and then recursively traverse all vertices adjacent to V.

b) If we perform this process on an arbitrary graph, we need to be careful to avoid cycles. To do this, when we visit a vertex 'V', we mark it visited, since now we have been there, and recursively call depth-first search on all adjacent vertices that are not already marked.

Template for Depth-First search

```
Void DFS( vertex V)
{
   visited [V] = True;
   for each W adjacent to V
   if( ! visited [W])
      DFS [W];
}
```

c) The global boolean array "visited [ ]" is initialized to false. By recursively calling the procedures only on nodes that have not been visited, we guarantee that we do not loop indefinetly.

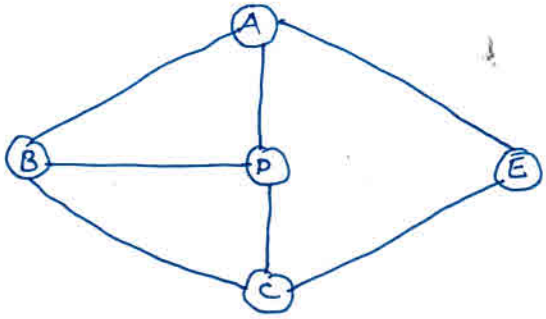
d) If the graph is undirected and not connected, or directed and not strongly connected, this strategy may fail to visit some nodes. we then search for an unmarked node, apply depth-first traversal there, and continue this process until there are no unmarked nodes.

Undirected Graphs:

a) An undirected graph is connected if and only if a depth-first search starting from any node visits every node.

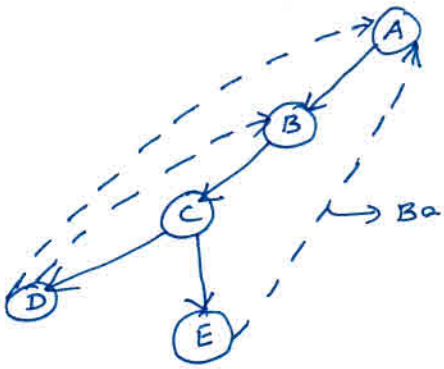
Example:

An Undirected Graph.



b) We start at Vertex A. Then we mark 'A' as visited and call Dfs(B) recursively. Dfs(B) marks B as visited and calls Dfs(C) recursively. Dfs(C) marks C as visited and calls Dfs(D) recursively. Dfs(D) sees both A and B, both these are already marked, so no recursive calls are made. Dfs(D) also sees that C is adjacent but marked, so no recursive call is made there, and Dfs(D) returns back to Dfs(C).

Dfs(C) sees B adjacent, ignores it, finds a previously unseen vertex E adjacent, and thus calls Dfs(E). Dfs(E) marks E, ignores A and C, and returns to Dfs(C). Dfs(C) returns to Dfs(B). Dfs(B) ignores both A and D and returns. Dfs(A) ignores both D and E and returns.



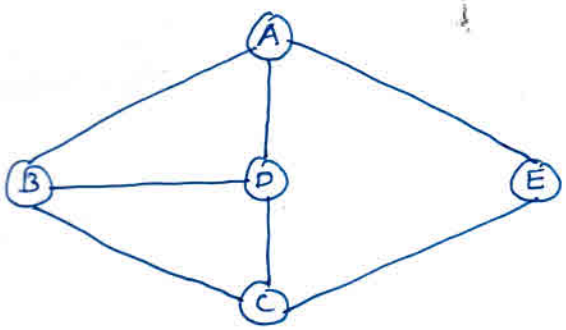
Depth first Spanning tree.

→ Back edge.

Biconnectivity:

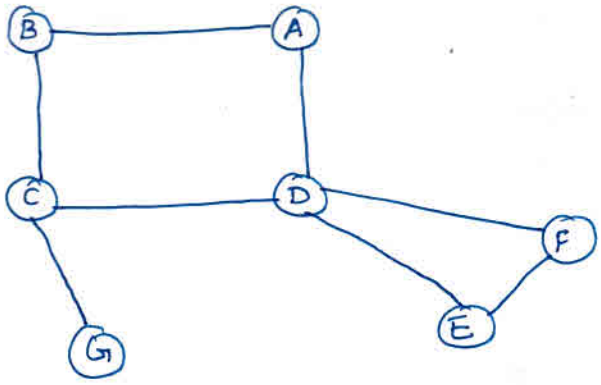
a) A Connected Undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.

Example: Graph which is Biconnected



b) If a graph is not biconnected, the vertices whose removal would disconnect the graph are known as articulation points.

Example: Graph which is not Biconnected



C, D are articulation Points. The removal of C would disconnect G, and the removal of D would disconnect E and F. from the rest of the graph.

c) Depth first search provides a linear time algorithm, to find all articulation points in a Connected graph.

Euler circuits

Example:-

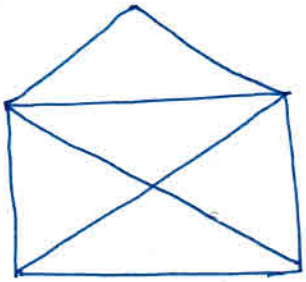


Diagram: 1

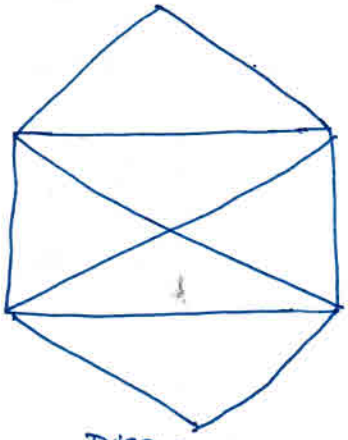


Diagram: 2

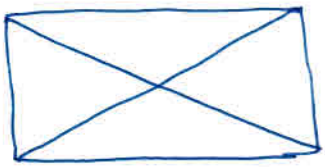


Diagram: 3

a) Consider above diagrams. A popular puzzle is to reconstruct these diagrams using a pen, drawing each line exactly once. The pen may not be lifted from the paper while the drawing is being performed. An extra challenge, make the pen finish at the same point at which it started.

b) The first diagram can be drawn only if the starting point is the lower left or right hand corner, and it is not possible to finish at the starting point. The second diagram is easily drawn with the finishing point the same as the starting point, but the third diagram cannot be drawn at all within the parameters of the puzzle.

c) we convert this problem to a graph theory problem by assigning a vertex to each intersection.

d) After this conversion is performed, we must find a path in the graph that visits every edge exactly once. If we want to solve the second challenge also, then we must find a cycle that visits every edge exactly once. This problem is commonly referred as Euler path (or Euler tour or

e) The First observation that can be made is that an Euler circuit, which must end on its starting vertex, is possible only if the graph is connected and each vertex has an even degree (number of Edges).

f) This is because, on the Euler Circuit, a vertex is entered and then left. If any vertex -v' has odd degree, then eventually we will reach the point where only one edge into v is unvisited, and taking it will strand us at v.

g) In any connected graph, all of whose vertices have even degree, must have an Euler circuit.