Unit-3

Greedy Method

Greedy Method

- Most straightforward design technique
  - Most problems have n inputs.
  - **Solution** contains a subset of inputs that **satisfies a given constraint**.
  - **Feasible solution**: Any subset that satisfies the constraint.
  - Need to find a feasible solution that ***maximizes or minimizes*** a given **objective function** – optimal solution.
- Used to determine a feasible solution that may or may not be optimal
  - At every point, make a decision that is locally optimal; and hope that it leads to a globally optimal solution.
  - Leads to a powerful method for getting a solution that works well for a wide range of applications.
  - May not guarantee the best solution.
- Ultimate goal is to find a ***feasible solution*** that ***minimizes [or maximizes] an objective function***; this solution is known as an optimal solution

Contd…

The greedy method can be used in 2 kinds of problems

Subset paradigm : need to find the optimal subset

Ordering paradigm: no need of optimal subset, we make decisions by considering the inputs in some order. Each decision is made using an optimization criterion that can be computed using decisions already made.

Ordering Paradigm	Subset Paradigm
Change Making	Huffman Coding
Container Loading	Optimal Storage on tapes
Machine Scheduling	Optimal merge patterns
Knapsack Problem	Single-source shortest paths
Tree vertex splitting	
Job Sequencing with deadlines	
Minimum-cost Spanning trees	

Greedy Algorithm(for subset paradigm)

```
Algorithm Greedy(a,n)
// a[1:n] contains the n inputs.
{
        solution:= Φ ; // Initalize the solution.
        for i:=1 to n do
        {
                x:= Select(a);
                if Feasible(solution,x) then
                        solution:= Union(solution,x);
        }
        return solution;
}
```

Select , Feasible and **Union** functions are implemented depending up on the given problem.

Huffman coding

- Suppose we have a 100,000-character data file that we wish to store compactly.

<table>
<tr><td>How to represent?</td><td colspan="2">Use a binary character coding (known as codeword) in which each character is represented by a unique binary string.</td></tr>
<tr><td>Find the alphabet i.e., what are the characters used and their frequencies?</td><td colspan="2">Assume characters used are a to f
<table>
<tr><th></th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th><th>f</th></tr>
<tr><td>Frequency(in thousands)</td><td>45</td><td>13</td><td>12</td><td>16</td><td>9</td><td>5</td></tr>
<tr><td>Fixed-Length codeword</td><td>000</td><td>001</td><td>010</td><td>011</td><td>100</td><td>101</td></tr>
<tr><td>Variable-length codeword</td><td>0</td><td>101</td><td>100</td><td>111</td><td>1101</td><td>1100</td></tr>
</table></td></tr>
<tr><td colspan="3">Storage required</td></tr>
<tr><td>Fixed-length codeword</td><td colspan="2">Total bits required = (45 *3 + 13 *3 + 12*3 +16 *3 + 9 * 3 +5 *3) *1000
= (45 + 13 + 12 + 16 + 9 + 5)*3*1000
= (100) *3000 = 300,000 bits</td></tr>
<tr><td>Variable-length codeword</td><td colspan="2">Total bits required = (45 *1 + 13 *3 + 12*3 +16 *3 + 9 * 4 +5 *4) *1000
= (45 + 39 + 36 + 48 + 36 + 20)*1000
= (224) *1000 = 224,000 bits</td></tr>
</table>

So , variable-length codeword is an optimal encoding which provides short codewords for frequent characters and long codewords for infrequent characters.

- We use ***prefix codes*** because it can always achieve ***optimal data compression*** among any other character code . A codeword is said to be **prefix code,** if no codeword is a prefix of any other codeword.

- A convenient representation of prefix code is ***binary tree***. where each binary codeword for a character is interpreted as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child."

- Given a tree T corresponding to a prefix code, we can easily **compute the number of bits** required to encode a file.

- For each character c in the alphabet C,

  let the attribute ***c.freq*** denote the

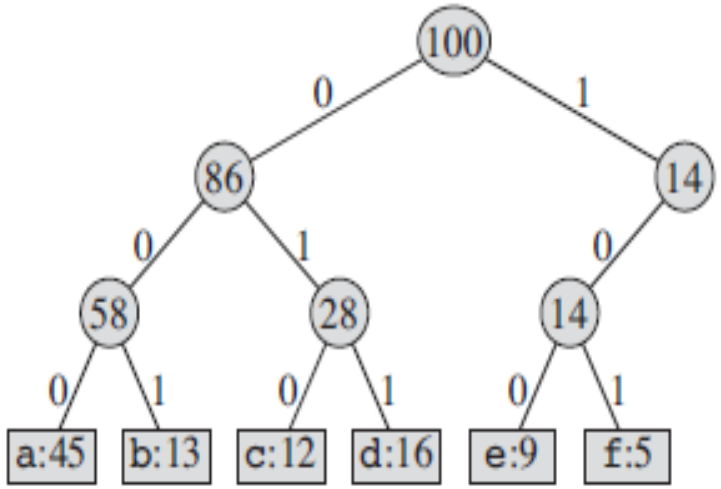
  frequency of c in the file and let $\mathbf{d_T(C)}$

  denote the depth of c's leaf in the tree.

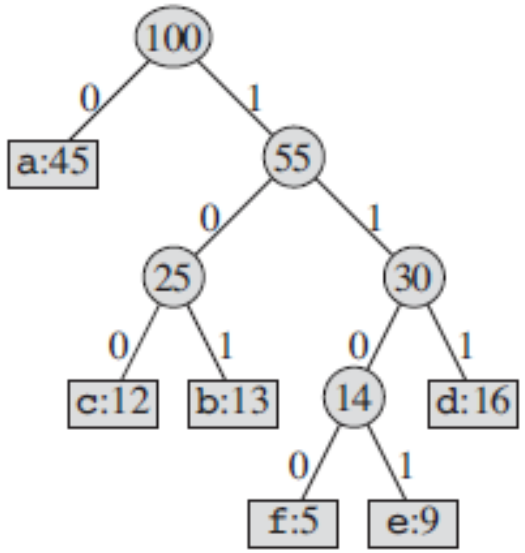
  The ***number of bits required*** to encode a

  file is

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$



Fixed-length codeword



Variable-length codeword

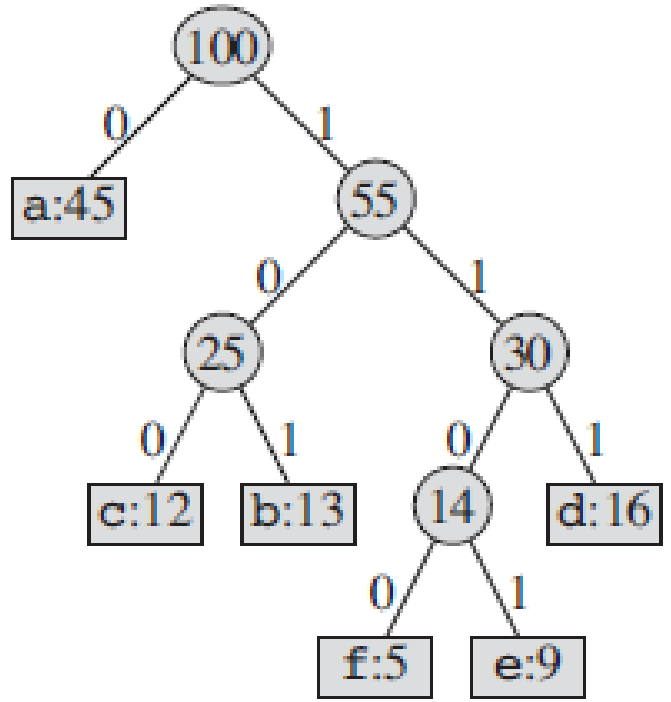
Encoding and Decoding of file

- ***Encoding*** is done by concatenating the codewords of each character of the file.

  Example – ***abc*** is encoded as 0.101.100 = 0101100

- ***Decoding*** requires a convenient representation for prefix code, which can determine unambiguously initial codewords . So, use a binary tree representation

  Example – the string ***001011101*** parses uniquely as
  0 . 0 . 101 . 1101, which decodes to ***aabe***.



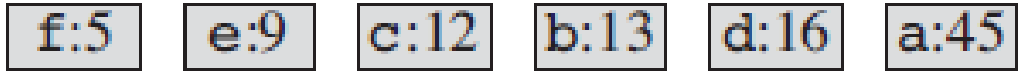
How to construct Huffman code?

Constructing a Huffman code

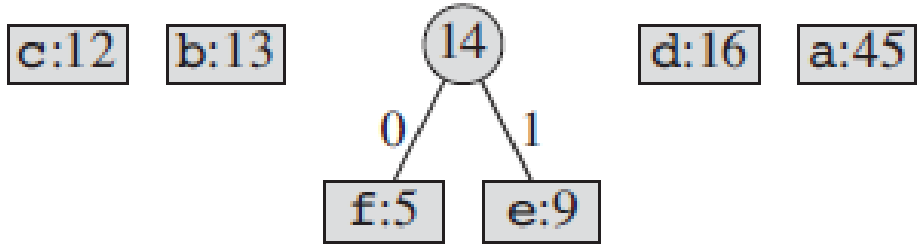
Huffman invented a greedy algorithm that constructs an optimal prefix code called a ***Huffman code***.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

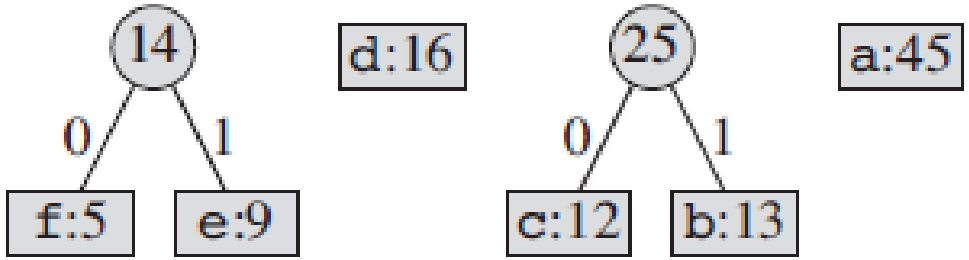
Step 1



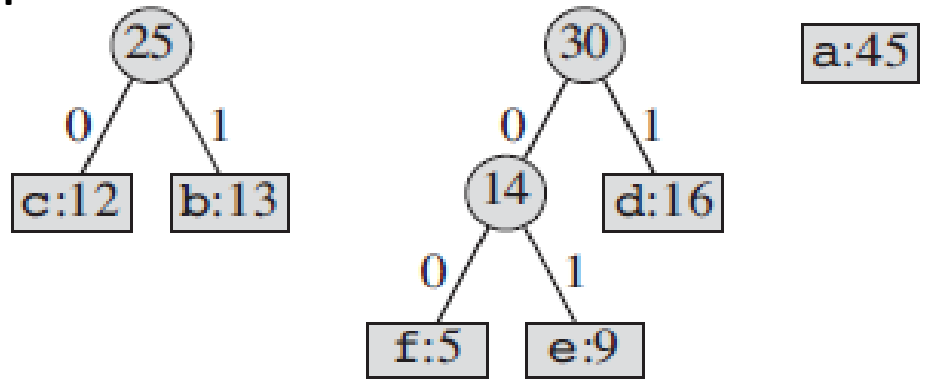
Step 2



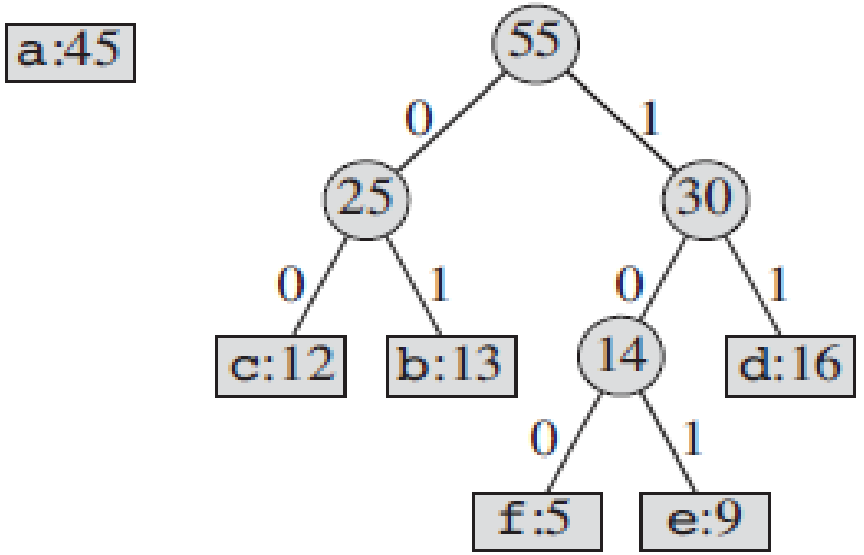
Step 3



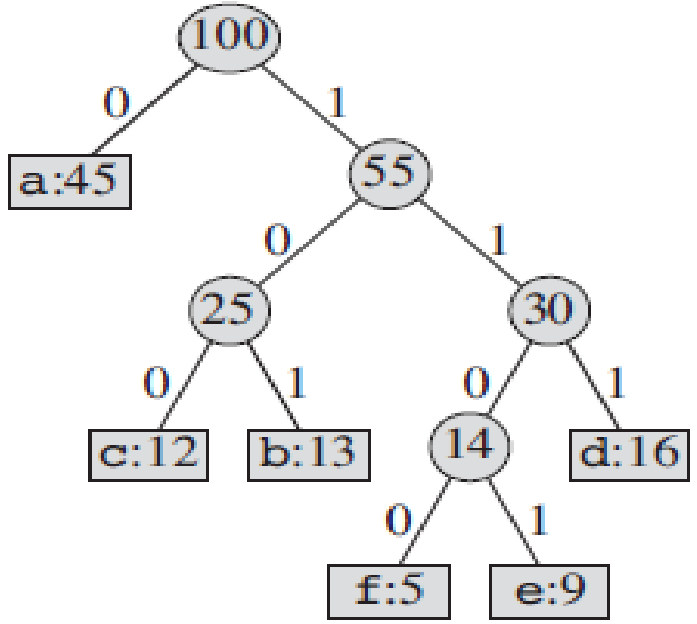
Step 4



Step 5



Step 6



Example

- Obtain a set of optimal Huffman codes for the seven messages (M1,M2,M3,M4,M5,M6,M7) with relative frequencies (q1,q2,q3,q4,q5,q6,q7)=(4,5,7,8,10,12,20) . Draw the decode tree for this set of codes.

Knapsack Problem(Fractional)

- Given n objects a knapsack or bag. Object *i* has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 < x_i < 1$,of object *i* is placed into the knapsack, then a profit $p_i x_i$ is earned. The ***objective*** is to obtain a filling of the knapsack that ***maximizes the total profit earned*** . Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

The profits and weights are positive numbers

- A feasible solution is any set ($x_1,x_2,....x_n$) satisfying $\sum_{1\le i\le n} w_i x_i \le m$ and

$$0 \le x_i \le 1, \quad 1 \le i \le n$$

- An optimal solution is a feasible solution that $\text{maximize} \sum_{1\le i\le n} p_i x_i$

Example

Consider the following instance of the knapsack problem n=3, m=20 , (p1,p2,p3)=(25,24,15) and (w1,w2,w3)=(18,15,10) .

S.No	(x1,x2,x3)	$\Sigma w_i x_i$	$\Sigma p_i x_i$
1	(1/2,1/3,1/4)	=18*1/2 +15*1/3+10*1/4 =9+5+2.5 =16.5	=25*1/2+24*1/3*15*1/4 =12.5+8+3.75 =24.25
2	(1,2/15,0)	20	28.2
3	(0,2/3,1)	20	31
4	(0,1,1/2)	20	31.5

Algorithm **GREEDY _KNAPSACK**(P, W, M, X, n)

```
// P[1:n] and W[1 :n] contain the profits and weights respectively of the n
//objects ordered so that P[i]/W[i] >= P[i + l]/W[i + 1].
//M is the knapsack size and X[1:n] is the solution vector
{
        X = 0 //initialize solution to zero
        rw = M // rw = remaining knapsack capacity
        for i = 1 to n do
        {
          if W[i] > rw then
                  break;
          X[i] = 1
          rw = rw - W[i]
        }
        if i <= n then
          X[i] = rw/W[i];
}
```

Example

- Consider the following instance of the knapsack problem: n = 5,
  (P1,P2,P3,P4,P5) =(30, 40, 45, 77, 90) , (W1,W2, W3, W4, W5) = (5,10,15,22,25) and m = 60

Job sequencing with deadlines

- Given a set of n jobs and one machine for processing jobs , and each job contains a deadline $d_i > 0$ (where d_i is an integer) and a profit $p_i > 0$. For any job *i* the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time.
- *The problem is to find an optimal subset of jobs that can be scheduled on a machine and all the jobs in the selected subset can be completed by its deadline.*
- A **feasible solution** for this problem is a **subset J of jobs** such that each job in this subset can be **completed by its deadline**.
- .The value of a **feasible solution J** is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$
- An **optimal solution** is a feasible solution with **maximum value**.

Example

Let n=4 , (p1,p2,p3,p4)= (100,10,15,27) and (d1,d2,d3,d4)= (2,1,2,1). The feasible solutions and their values are

S. No	Possibilities	Feasible ?
1	{1}	Yes
2	{2}	Yes
3	{3}	Yes
4	{4}	Yes
5	{1, 2}	Yes
6	{1, 3}	Yes
7	{1, 4}	Yes
8	{2, 3}	Yes
9	{2, 4}	No
10	{3, 4}	Yes
11	{1, 2, 3}	No
12	{1, 2, 4}	No
13	{1, 3, 4}	No
14	{2, 3, 4}	No
15	{1, 2, 3, 4}	No

machine

slot1	slot2

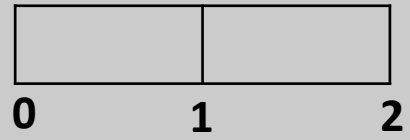
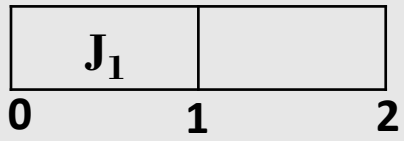
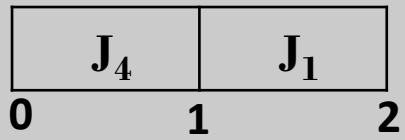
0 1 2

S.No	Feasible Solution	Processing Sequence	Value
1	(1,2)	2,1	110
2	(1,3)	1,3 or 3,1	115
3	**(1,4)**	**4,1**	**127**
4	(2,3)	2,3	25
5	(3,4)	4,3	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

Greedy Solution to JS with deadlines

Let n=4 , (p1,p2,p3,p4)= (100,10,15,27) and (d1,d2,d3,d4)= (2,1,2,1).

- Arrange the jobs in the decreasing (non-increasing) order of their profits. So, the job selection order is (J_1, J_4, J_3, J_2)
- Number of slots of the machine = min(n , max(deadline of jobs)) = min(4, 2) = 2

S.No	Job Selected	Slot allotted	Slots Available	Profit
				0
1	J_1	Is feasible? Yes Allot : [0,1]		100
2	J_4	Is feasible? Yes Move the job J_1 to $[1,2]$ slot and allot [0,1] to job J_4		127

```
1 Algorithm JS(d,j,n)
2 // d[i]>=1,1<=i<=n arethe deadlines, n>=1.The jobs are ordered such that p[1] >=p[2]>=.....>=p[n].
3 // J[i] is the ith job in the optimal solution,1<=i<=k. Also, at termination d[J[i]]<d[J[i+ 1]], 1<=i<k.
4 {
5         d[0]:=J[0]:=0;        // Initialize.
6         J[1]:=1;              //  Include job1.
7         k:=1;
8         for i :=2 to n do
9         {
10        // Consider jobs in nonincreasing order of p[i]. Find position for i and check feasibility of insertion.
11                r :=k;
12                while ((d[J[r]]>d[i]) and (d[J[r]]!= r)) do r :=r -1;
13                if ((d[J[r]]<=d[i])and (d[i] >r)) then
14                {
15                        // Insert i into J[].
16                        for q :=k to (r + 1) step -1 do
17                                J[q+ 1]:=J[q];
18                        J[r+ 1]:= i; k :=k + 1;
19                }
20        }
21        return k;
22 }
```

Summary

Name of the Problem	Brute force solution	Optimization Measure	Time Complexity reduced to
Huffman coding	In what order we need to merge the frequencies. **Possibilities : n!**	Select the minimum two frequencies from the priority queue.	O(n!) ------> O(n log n)
Fractional Knapsack	Which subset of the objects leads to maximum profit **Possibilities : 2^n**	Select the objects based on the P/W ratio of object(first largest ,second largest and so)	O(2^n) -------> O(n log n)
Job Sequencing with deadlines	Which subset of the jobs that can be completed by its deadline leads to maximum profit **Possibilities : 2^n**	Select the jobs based on the profit (first largest ,second largest and so) and check the feasibility	O(2^n) -------> O(n^2)

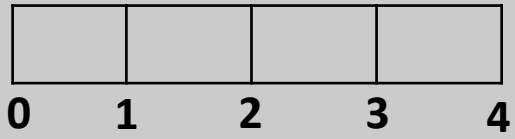
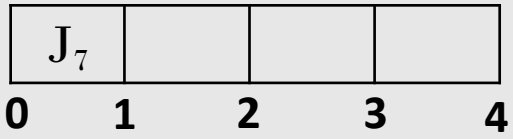
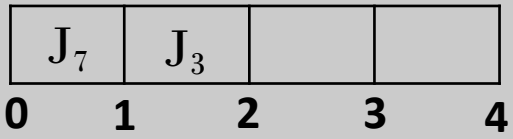
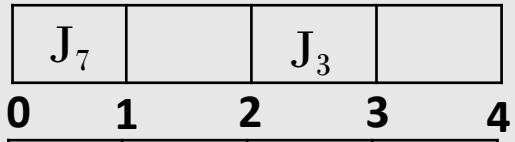
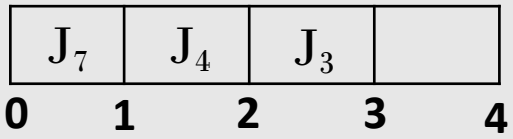
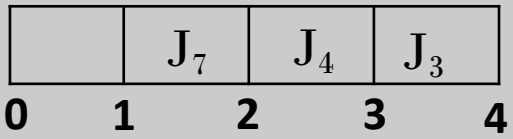
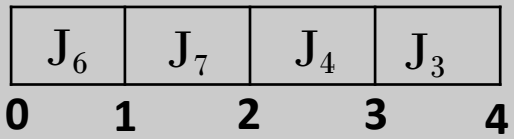
Job sequencing problem

What is the solution generated by the function JS when n =7,

(P1,P2,P3,P4,P5,P6,P7)= (3,5,20,18,1,6,30)

(d1,d2,d3,d4,d5,d6,d7) = (1,3,4,3,2,1,2)

(P1,P2,P3,P4,P5,P6,P7)= (3,5,20,18,1,6,30) (d1,d2,d3,d4,d5,d6,d7) = (1,3,4,3,2,1,2)

S.No	Job Selected	Slot allotted	Slots Available	Profit
				0
1	J_7	Is feasible? Yes Allot : [0,1]		30
2	J_3	Is feasible? Yes Allot : [1,2]		50
3	J_4	Is feasible? Yes Move the job J_3 to [2,3] slot and allot [1,2] to job J_4	 	68
4	J_6	Is feasible? Yes Move the job J_3 to [3,4], J_4 to [2,3] , J_7 to [1,2] slot and allot [0,1] to job J_6	 	72

Example

Let n=4 , (p1,p2,p3,p4)= (100,10,15,27) and (d1,d2,d3,d4)= (2,1,2,1). The feasible solutions and their values are

S.No	Feasible Solution	Processing Sequence	Value
1	(1,2)	2,1	110
2	(1,3)	1,3 or 3,1	115
3	(1,4)	4,1	127
4	(2,3)	2,3	25
5	(3,4)	4,3	42
6	(1)	1	100
7	(2)	2	10
8	(3)	3	15
9	(4)	4	27

Minimum Spanning tree

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, we can use an arrangement of n-1 wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

We can model this wiring problem with a connected, undirected graph G=(V, E), where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge (u , v) ϵ E, we have a weight w (u , v) ; specifying the cost (amount of wire needed) to connect u and v. We then wish to find an acyclic subset T $\subseteq$ E that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a ***spanning tree*** since it "spans" the graph G. We call the problem of determining the tree T the ***minimum-spanning-tree problem***.

Properties of Spanning tree

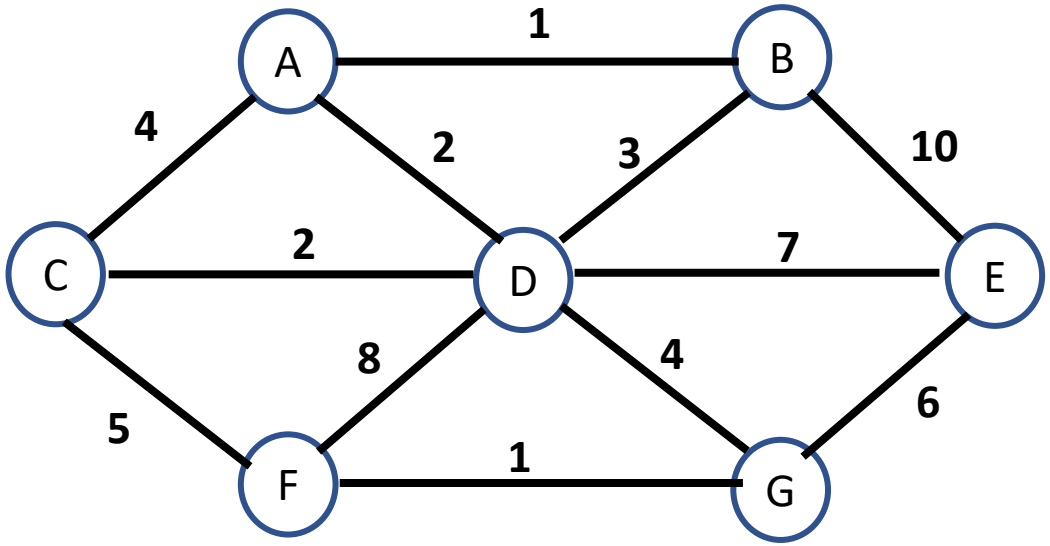
- In a spanning tree, the number of edges will always be |V|- 1
- A Spanning tree doesn't contain any loops or cycles
- If we remove any edge from the spanning tree, then it will be disconnected.
- It we add one edge to a spanning tree , then it will create a cycle.
- The number of spanning trees of a complete graph is $V^{|v|-2}$. (Where |V| is number of vertices of a graph .

Algorithms for finding Minimum spanning Tree

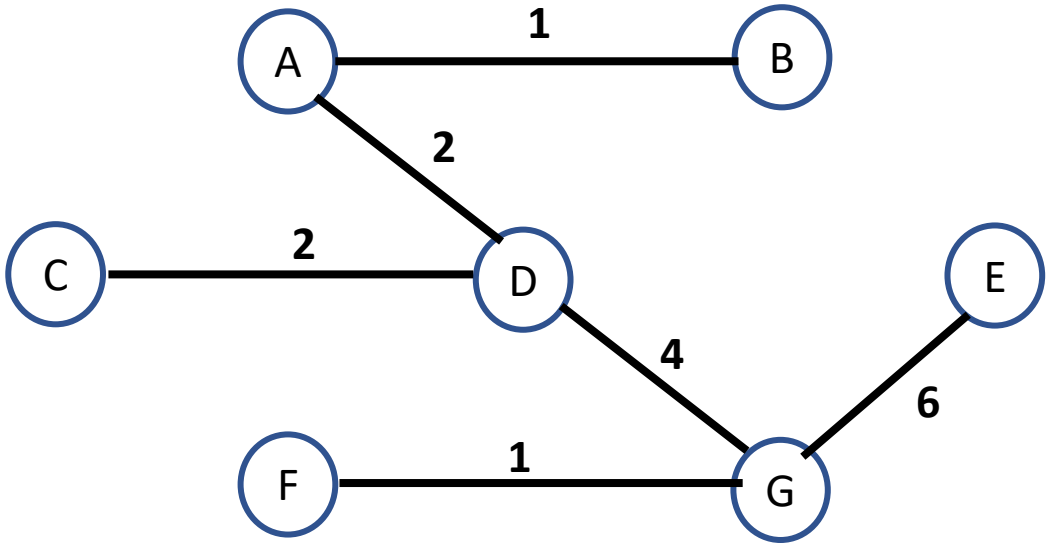
- Prim’s Algorithm
- Kruskal’s Algorithm

Prim's Algorithm

```
Algorithm Prim (G,s)
// G is graph that contains adjacency information and
// s is the start vertex.
{
1.      T = { } // includes tree edges
2.      TV = {s}; // Includes tree vertices , initialized to start vertex s
3.      while ( T contains fewer than n-1 edges )
4.      {
5.              Let (u, v) be a least cost edge such that u ∈ TV and v ∉ TV
6.              if (there is no such edge)
7.                      break;
8.              add v to TV;
9.              add (u,v) to T;
10.     }
11.      if ( T contains fewer than n-1 edges)
12.             printf(" No spanning tree \n");
}
```



Graph



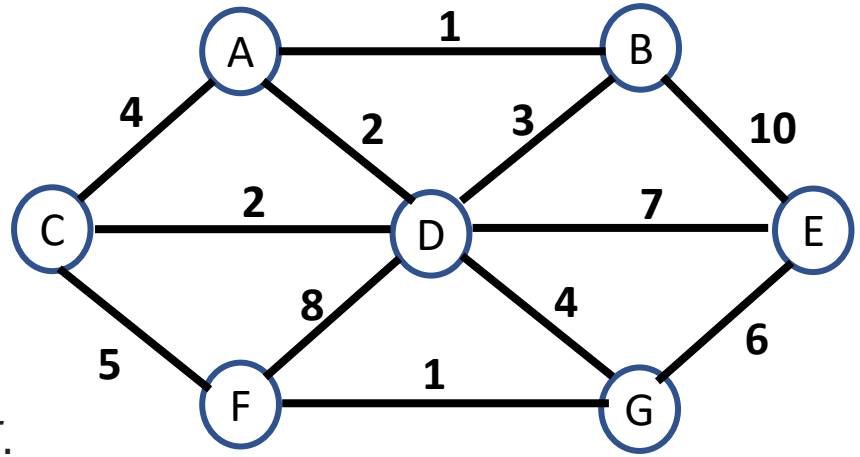
Minimum Spanning Tree

Tree edges
T= { }

Tree vertices
TV = { A }

Iteration 1

1. Find the neighbors of TV ={A}
   A : B, C, D
2. Consider the neighbors of step1 ∉ TV and find min edge.
   i.e., min{ **<A,B>=1**, <A,C>=4, <A,D>=2 }
3. Min edge is <A, B> then include *vertex B in TV* and edge *<A,B> in T*.
   T= { <A,B>} TV = {A , B}



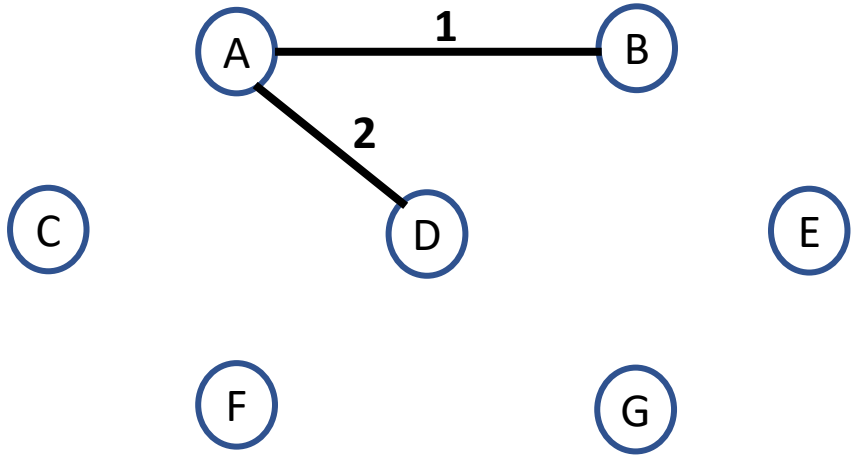
Graph

Iteration 2

Tree edges
T= { <A,B> }

Tree vertices
TV = { A, B }

1. Find the neighbors of TV ={A,B}
   A : ~~B~~, C, D
   B : ~~A~~, D, E
2. Consider the neighbors of step1 ∉ TV and find min edge.
   i.e., min{ <A, C>=4, **<A, D>=2**, <B, D> =3, <B, E>= 10 }
3. Min edge is <A,D> then include *vertex D in TV* and edge <A,D> in T.
   T= { <A,B>, <A,D> } TV = {A , B, D}



Minimum spanning tree

Iteration 3

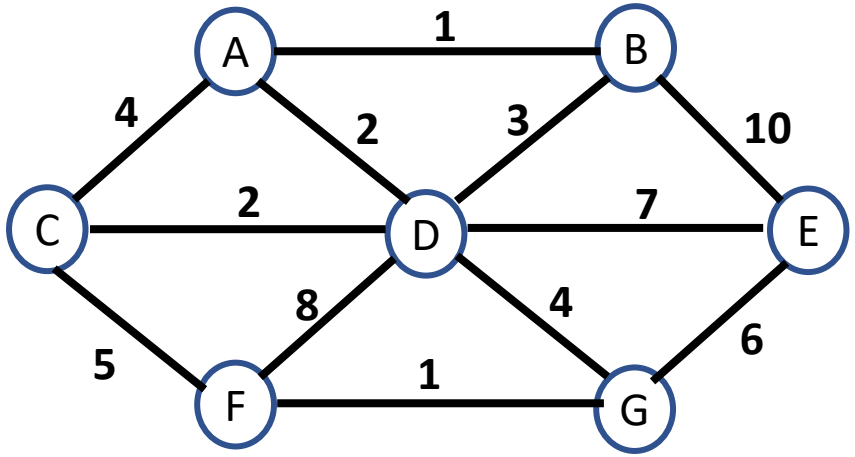
Tree edges
T= { <A,B> ,<A,D>}

Tree vertices
TV = { A, B,D }

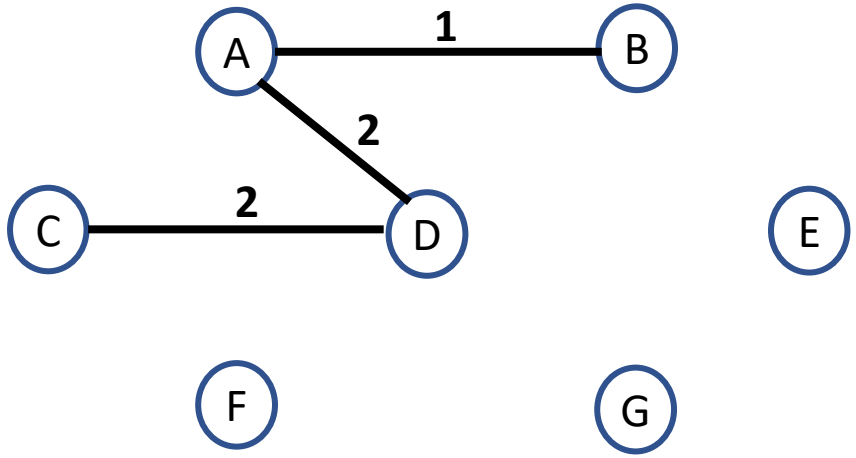
1. Find the neighbors of TV ={A, B, D}
   A : ~~B~~, C, ~~D~~
   B : ~~A~~, ~~D~~, E
   D : ~~A~~, ~~B~~, C, E, F, G

2. Consider the neighbors of step1 ∉ TV and find min edge.
   i.e., min{ <A, C> = 4, <B, E> = 10,
   <D, C> = 2,<D, E> = 7 , <D, F> = 8, <D, G> = 4
   }

3. Min edge is <D, C> then include *vertex C in TV* and edge <D,C> in T.
   T= { <A,B>, <A,D> ,<D, C> } TV = {A , B, D, C}



Graph



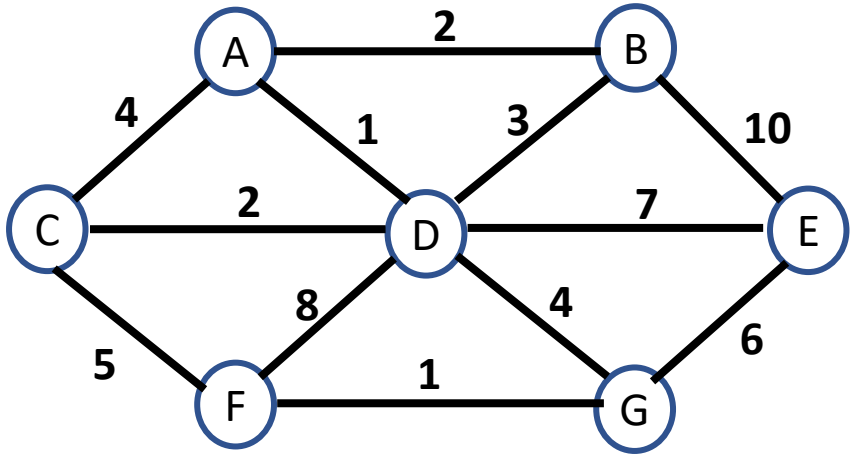
Minimum spanning tree

Iteration 4

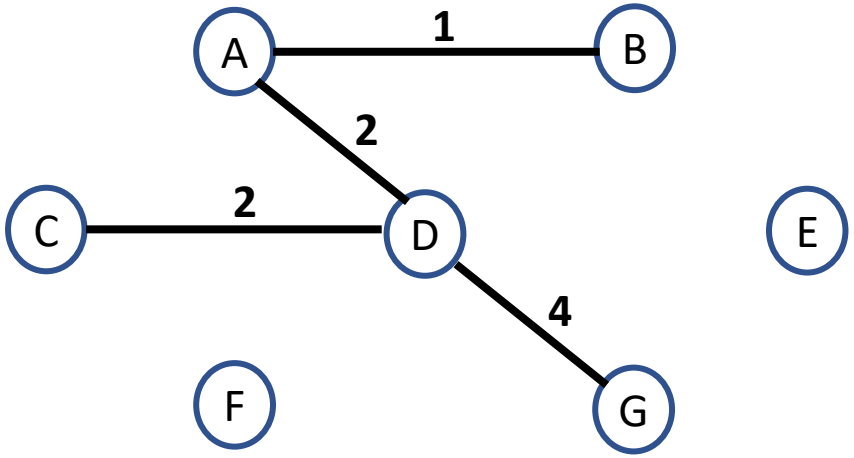
Tree edges
T= { <A,B>, <A,D> ,<D, C> }

Tree vertices
TV = { A, B, D, C }

1. Find the neighbors of TV ={A, B, D, C}
   A : ~~B~~, ~~C~~, ~~D~~
   B : ~~A~~, ~~D~~, E
   D : ~~A~~, ~~B~~, ~~C~~, E, F, G
   C: ~~A~~, ~~D~~, F
2. Consider the neighbors of step1 ∉ TV and find min edge.
   i.e., min{ <B, E> = 10,
   <D, E> = 7 , <D, F> = 8, **<D, G> = 4**,
   <C, F> = 5
   }
3. Min edge is <D, G> then include *vertex G in TV* and edge <D,G> in T.
   T= { <A,B>, <A,D> , <D, C>, <D, G> } TV = {A , B, D, C, G}



Graph



Minimum spanning tree

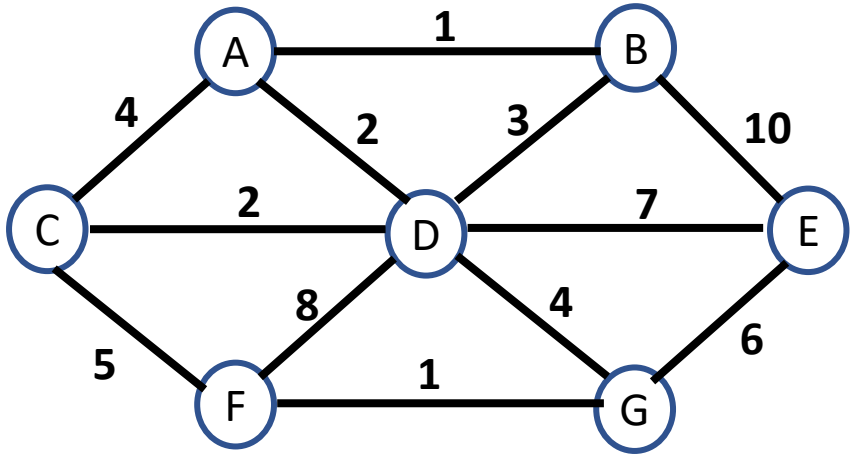
Iteration 5

Tree edges

T= { <A,B>, <A,D> ,<D, C>,<D, G> }

Tree vertices

TV = { A, B, D, C, G }



Graph

1. Find the neighbors of TV ={A, B, D, C, G}

   A : ~~B~~, ~~C~~, ~~D~~

   B : ~~A~~, ~~D~~, E

   D : ~~A~~, ~~B~~, ~~C~~, E, F, ~~G~~

   C: ~~A~~, ~~D~~, F

   G: ~~D~~, F, E

2. Consider the neighbors of step1 ∉ TV and find min edge.

   i.e., min{ <B, E> = 10,

   <D, E> = 7 , <D, F> = 8,

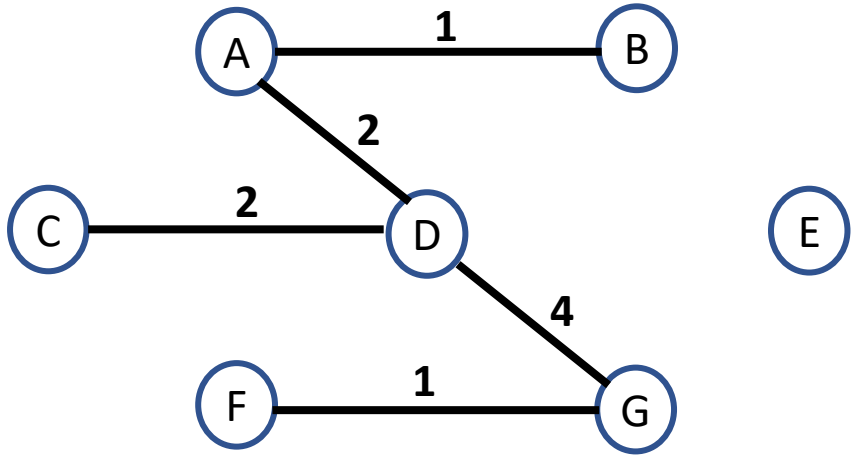
   <C, F> = 5

   <G, F> = 1 , <G, E> = 6

   }

3. Min edge is <G, F> then include *vertex F in TV* and edge <G, F> in T.

   T= { <A,B>, <A,D> , <D, C>, <D, G>,<G, F> } TV = {A , B, D, C, G, F}



Minimum spanning tree

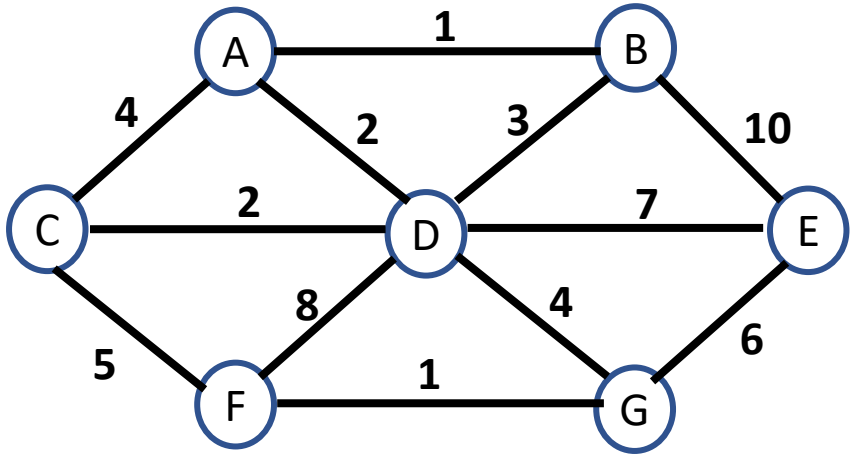
Iteration 6

Tree edges

T= { <A,B>, <A,D> , <D, C>, <D, G>, <G, F> }

Tree vertices

TV = { A, B, D, C, G, F }



Graph

1. Find the neighbors of TV ={A, B, D, C, G, F}

   A : ~~B, C, D~~

   B : ~~A, D~~, E

   D : ~~A, B, C~~, E, ~~F, G~~

   C: ~~A, D~~, F

   G: ~~D, F~~, E

   F: ~~C, D, G~~

2. Consider the neighbors of step1 ∉ TV and find min edge.

   i.e., min{ <B, E> = 10,

   <D, E> = 7 ,

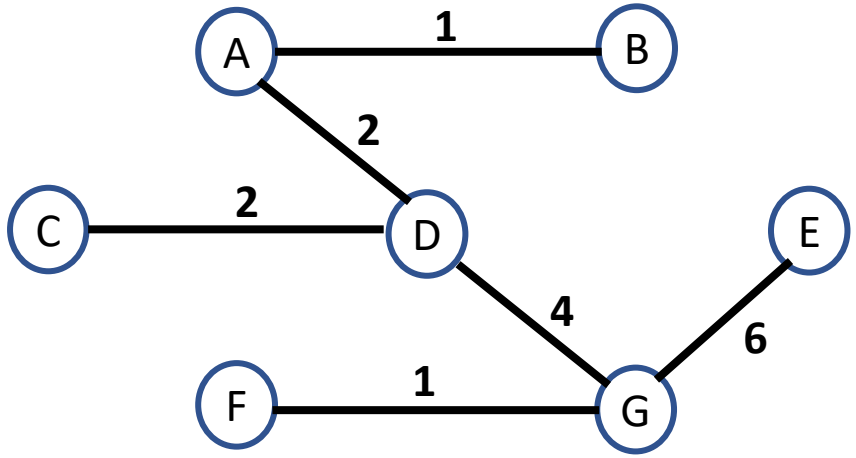
   <G, E> = 6

   }

3. Min edge is <G, E> then include *vertex E in TV* and edge <G, E> in T.

   T= { <A,B>, <A,D> , <D, C>, <D, G>,<G, F> , <G, E>}

   TV = {A , B, D, C, G, F, E}



Minimum spanning tree

Kruskal's Algorithm

```
Algorithm Kruskal(G)
{
1.      T = {}; // includes tree edges
2.      while ( T Contains less than n-1 edges && E is not empty)
3.      {
4.              choose a least cost edge (v,w) from E;
5.              delete (v,w) from E;
6.              if ( (v, w) does not create a cycle in T)
7.                      add (v, w) to T;
8.              else
9.                      discard (v,w);
10.     }
11.     if ( T contains fewer than n-1 edges)
12.             printf( " No spanning tree\n");
```

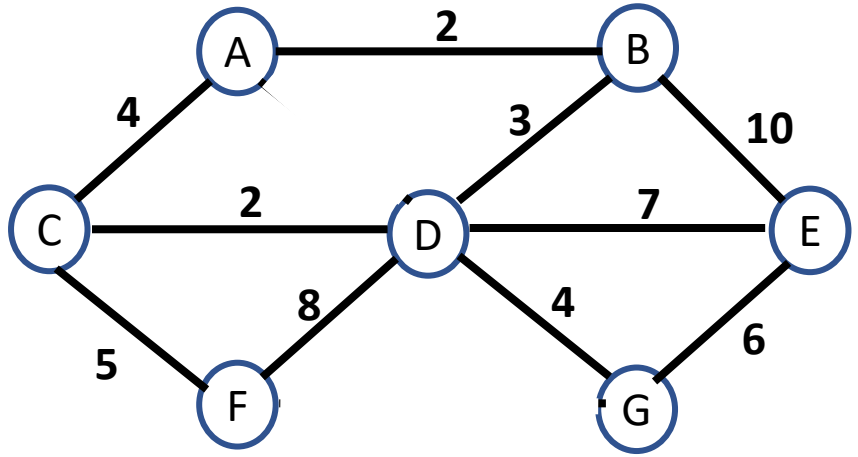
Tree edges

T= { }

Iteration 1

1. Choose least cost edge <u , v> from the graph G
   <A, D> = 1 (or <F, G> = 1)
2. Delete <A,D> from E
3. Is <A,D> create a cycle in T ? No .
   add <A,D> to T

T= { <A, D> }



Graph

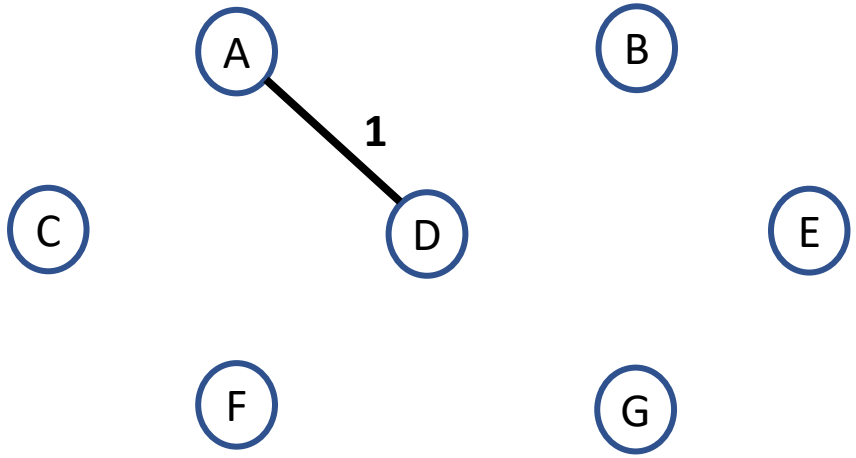
Iteration 2

Tree edges

T= { <A,D> }

1. Choose least cost edge <u , v> from the graph G
   <F, G> = 1
2. Delete <F, G> from E
3. Is <F,G> create a cycle in T? No
   add <F, G> to T

T= { <A, D> , <F, G> }



Minimum spanning tree

Tree edges

T= {<A, D> , <F, G> }

Iteration 3

1. Choose least cost edge <u , v> from the graph G
   <C, D> = 2 (or <A, B> = 2)
2. Delete <C, D> from E
3. Is <C,D> create a cycle in T ? No
   add <C, D> to T

T= { <A, D>, <F, G>, <C, D>}

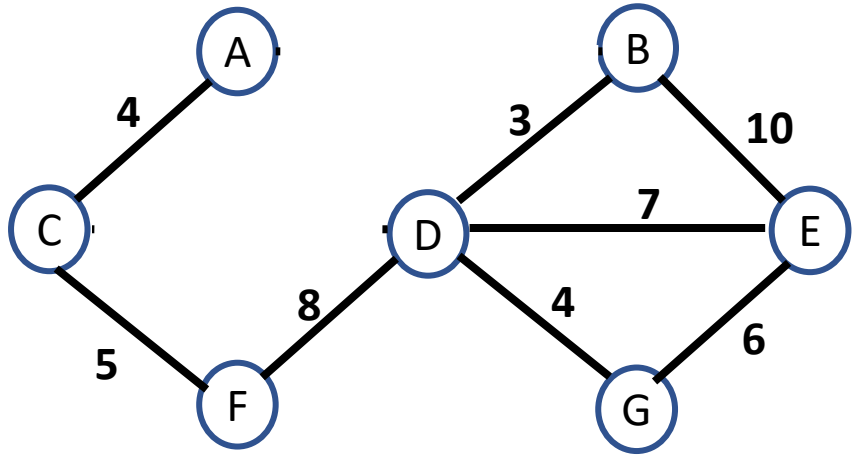
Iteration 4

Tree edges

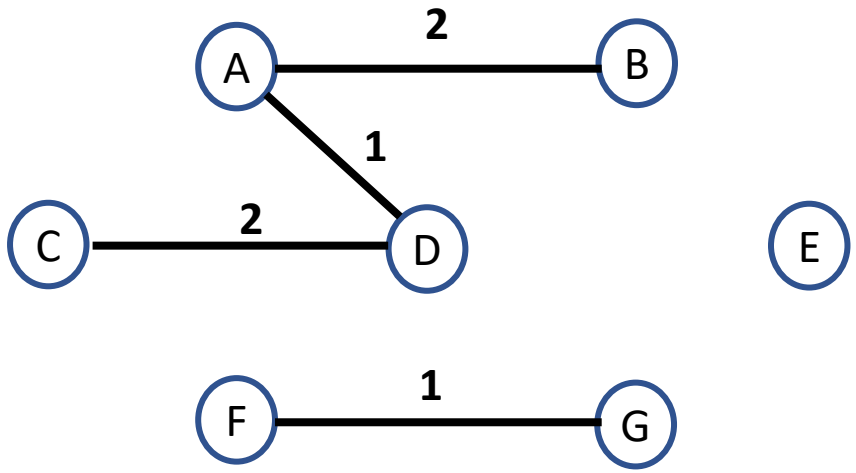
T= { <A, D>, <F, G>, <C, D>}

1. Choose least cost edge <u , v> from the graph G
   <A, B> = 2
2. Delete <A, B> from E
3. Is <A, B> create a cycle in T ? No
   add <A, B> to T

T= { <A, D> , <F, G>, <C, D>, <A, B>}



Graph



Minimum spanning tree

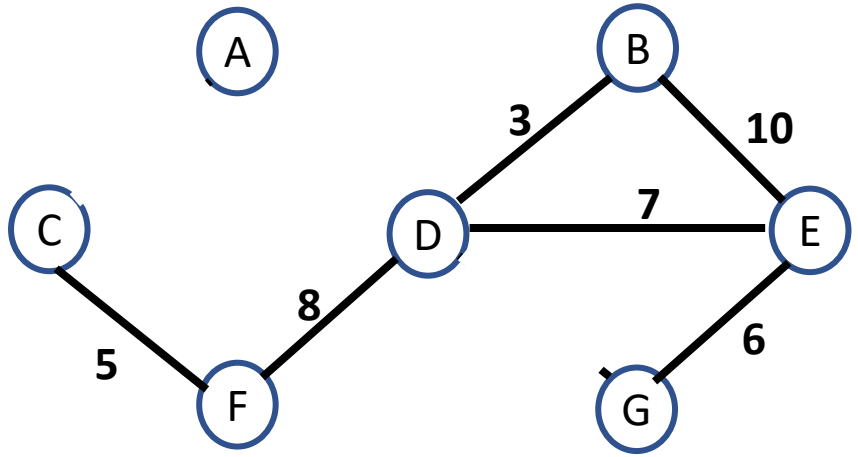
Tree edges

T= { <A, D> , <F, G>, <C, D>, <A, B>}

Iteration 5

1. Choose least cost edge <u , v> from the graph G
   <A, C> = 4 (or <D, G> = 4)
2. Delete <A, C> from E
3. Is <A, C> creates a cycle in T ? Yes
   discard <A, C> to T

T= { <A, D>, <F, G>, <C, D>,<A, B>}



Graph

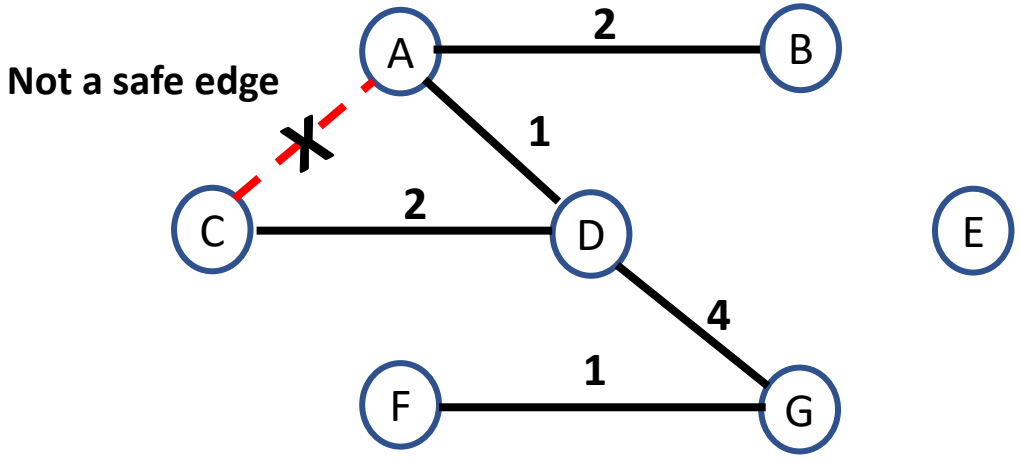
Iteration 6

Tree edges

T= { <A, D>, <F, G>, <C, D>, <A, B>}

1. Choose least cost edge <u , v> from the graph G
   <D, G> = 4
2. Delete <D, G> from E
3. Is <D, G> create a cycle in T ? No
   add <D, G> to T

T= { <A, D> , <F, G>, <C, D>, <A, B> , <D, G>}



Minimum spanning tree

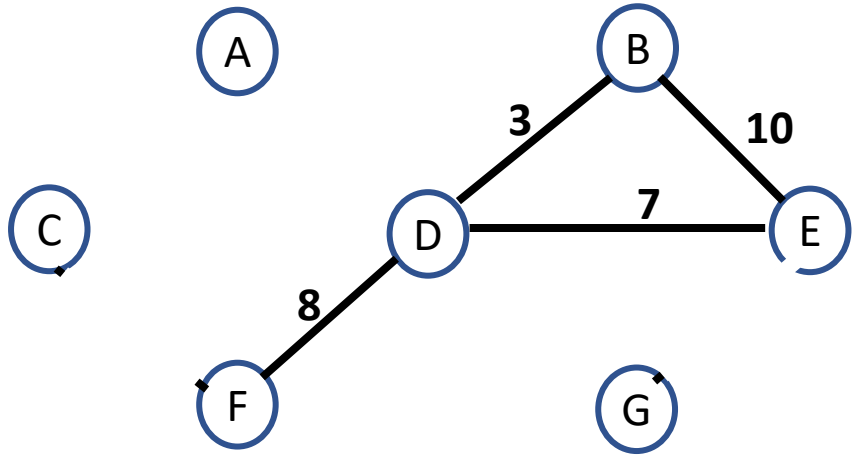
Tree edges

T= { <A, D> , <F, G>, <C, D>, <A, B>, <D, G> }

Iteration 7

1. Choose least cost edge <u , v> from the graph G
   <C, F> = 5
2. Delete <C, F> from E
3. Is <C, F> creates a cycle in T ? yes
   discard <C, F> to T

T= { <A, D>, <F, G>, <C, D>,<A, B>, <D, G>}



Graph

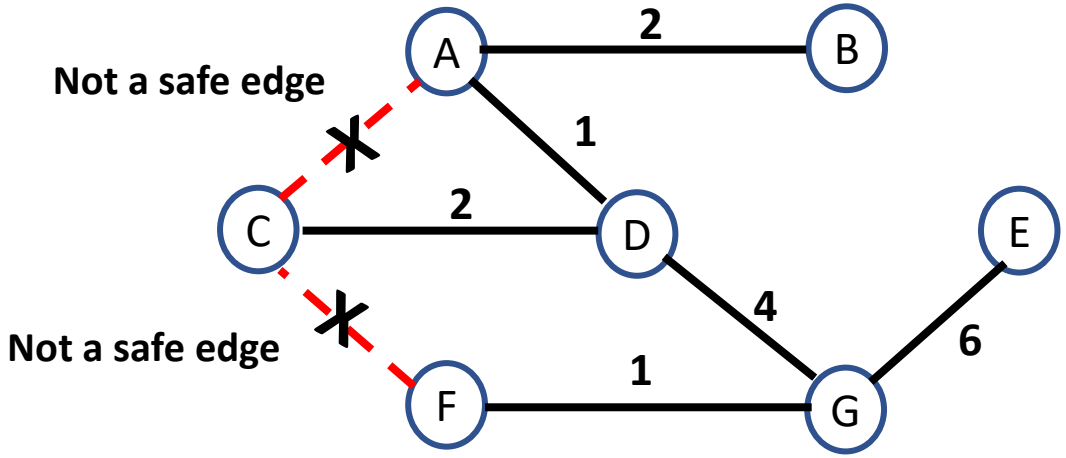
Iteration 8

Tree edges

T= { <A, D>, <F, G>, <C, D>, <A, B>, <D, G>}

1. Choose least cost edge <u , v> from the graph G
   <E, G> = 6
2. Delete <E, G> from E
3. Is <E, G> create a cycle in T ? No
   add <E, G> to T

T= { <A, D> , <F, G>, <C, D>, <A, B> , <D, G>, <E, G>}



Minimum spanning tree

```
1.Algorithm Prim(int E[][Size],float cost[][size],int n,int t[][2]){
2.          int near[size],j,k,l;
3.          let (k , l) be an edge of minimum cost in E;
4.          float mincost=cost[k][l];
5.          t[1][1]=k ; t[1][2]=l;
6.          for(int i=1;i<=n;i++){
7.                      if(cost[i][l]<cost[i][k]) near[i]=l
8.                      else near[i]=k;
9.          }
10.         near[k]=near[l]=0;
11.         for(i=2;i<=n-1;i++) // find n-2 additional edges for t.
12.         {
13.                     let j be an index such that near[j]!=0 and cost[j,near[j]] is minimum;
14.                     t[i][1]=j;t[i][2]=near[j];
15.                     mincost=mincost+cost[j][near[j]];
16.                     near[j]=0;
17.                     for(k=1;k<=n;k++) // update near[]
18.                     {
19.                                 if(near[k]!=0 && (cost[k][near[k]]>cost[k][j]))
20.                                             near[k]=j;
21.                     }
22.         }
23.         return mincost;
24. }
```

O(|E|)

Θ (n)

O(n)

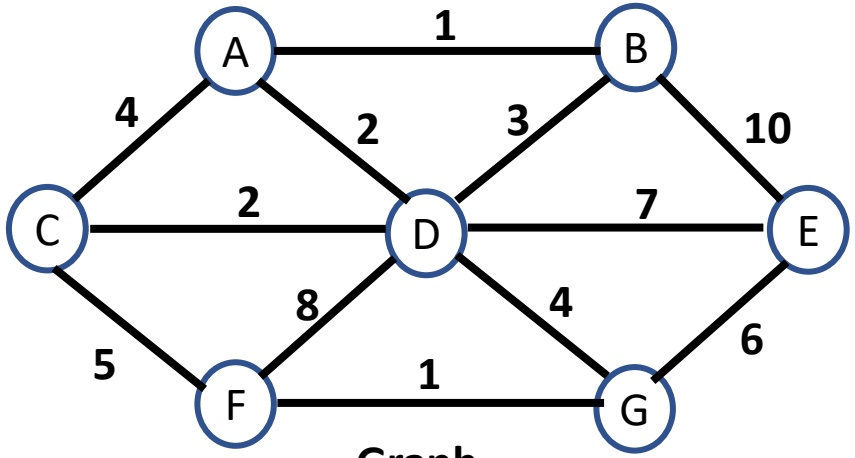
O(n)

$O(n^2)$

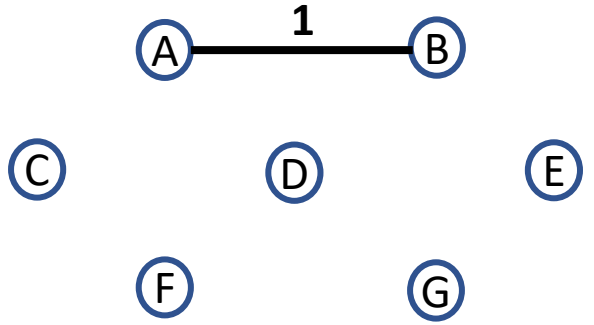
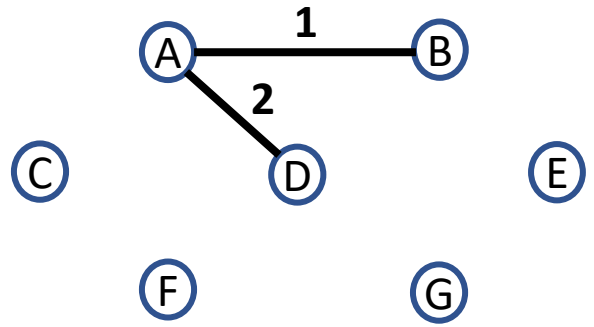
$O(n+n+n^2) = O(n^2)$

Initial step

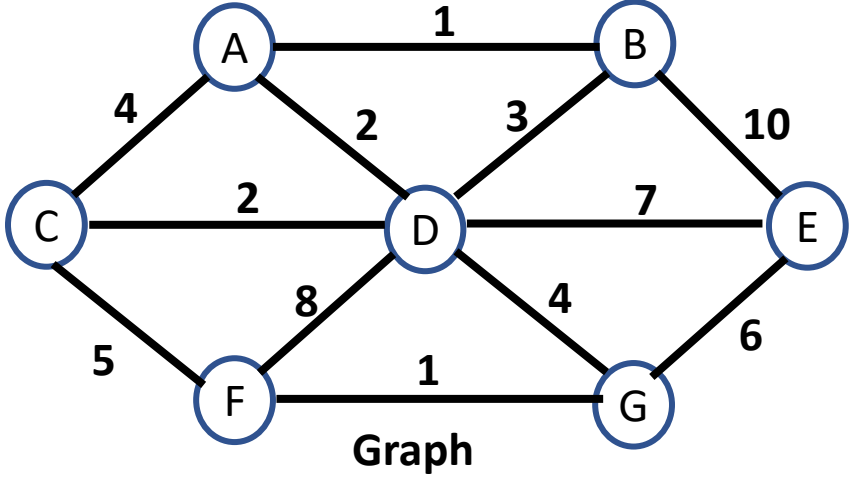
mincost =1



Graph

	Near array	Minimum Spanning tree
Initial step • Let (k , l) be a minimum cost edge in the graph i.e., <A , B> = 1 • find the near of each node • Mark near[A] and near[B] • Tree edges = { <A,B> }	<table><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th><th>G</th></tr><tr><td>near</td><td>~~A~~</td><td>~~A~~</td><td>A</td><td>A</td><td>B</td><td>A</td><td>A</td></tr><tr><td>Cost</td><td>-</td><td>-</td><td>4</td><td>2</td><td>10</td><td>∞</td><td>∞</td></tr></table>mincost =1	
• Find a min cost node j such that cost[j][near[j]] is minimum. • i.e., D and the edge is <D,A> = 2 • Include the <A,D> in tree edges. Tree edges = {<A,B>, <D,A>} • Update Near array	<table><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th><th>G</th></tr><tr><td>near</td><td>~~A~~</td><td>~~A~~</td><td>~~A~~ D</td><td>~~A~~</td><td>~~B~~ D</td><td>~~A~~D</td><td>~~A~~ D</td></tr><tr><td>Cost</td><td>-</td><td>-</td><td>~~4~~ 2</td><td>-</td><td>~~10~~7</td><td>~~∞~~8</td><td>~~∞~~ 4</td></tr></table>mincost =1+2=3	

	A	B	C	D	E	F	G
near	~~A~~	~~A~~	D	~~A~~	D	D	D
Cost	-	-	2	-	7	8	4



Graph

	Near array	Minimum Spanning tree
• Find a min cost node j such that cost[j][near[j]] is minimum. i.e., C and the edge is <C,D> = 2 • Include the <C,D> in tree edges. Tree edges = {<A,B>, <A,D>, <C,D>} • Update Near array		
• Find a min cost node j such that cost[j][near[j]] is minimum. • i.e., G and the edge is <G,D> = 4 • Include the <C,D> in tree edges. • Tree edges = {<A,B>, <A,D>, <C,D>, <G,D>} • Update Near array		

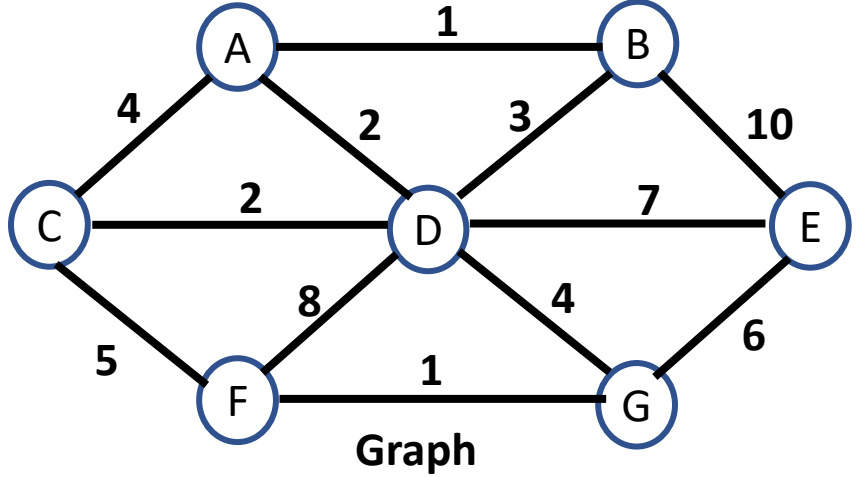
	A	B	C	D	E	F	G
near	~~A~~	~~A~~	~~D~~	~~A~~	D	~~D~~ C	D
Cost	-	-	~~2~~	-	7	~~8~~ 5	4

mincost =3+2=5

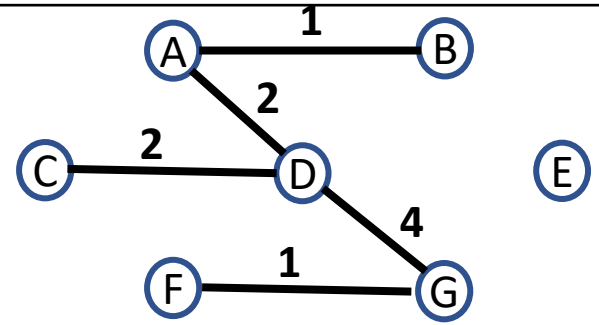
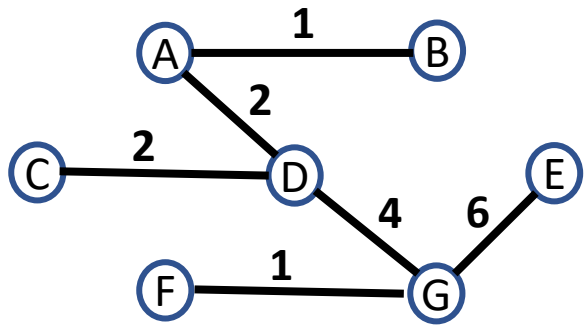
	A	B	C	D	E	F	G
near	~~A~~	~~A~~	~~D~~	~~A~~	~~D~~ G	~~C~~ G	~~D~~
Cost	-	-	-	-	~~7~~ 6	~~5~~ 1	~~4~~

mincost =5+4=9

	A	B	C	D	E	F	G
near	~~A~~	~~A~~	~~D~~	~~A~~	G	G	~~D~~
Cost	-	-	-	-	6	1	-



Graph

	Near array	Minimum Spanning tree
• Find a min cost node j such that cost[j][near[j]] is minimum. i.e., F and the edge is <F,G> = 1 • Include the <F,G> in tree edges. • Tree edges = {<A,B>, <D,A>, <C,D>, <G,D>,<F,G>} • Update Near array		
• Find a min cost node j such that cost[j][near[j]] is minimum. • i.e., E and the edge is <E,G> = 6 • Include the <E,G> in tree edges. • Tree edges = {<A,B>, <D,A>, <C,D>, <G,D>, <F,G>,<E,G>}		

	A	B	C	D	E	F	G
near	~~A~~	~~A~~	~~D~~	~~A~~	G	~~G~~	~~D~~
Cost	-	-	-	-	6	~~1~~	-

mincost =9+1=10

	A	B	C	D	E	F	G
near	~~A~~	~~A~~	~~D~~	~~A~~	~~G~~	~~G~~	~~D~~
Cost	-	-	-	-	~~6~~	-	-

mincost =10+6=16

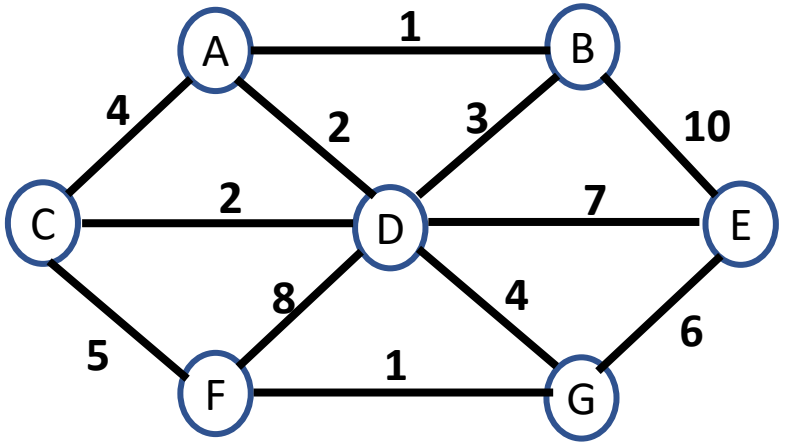
```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

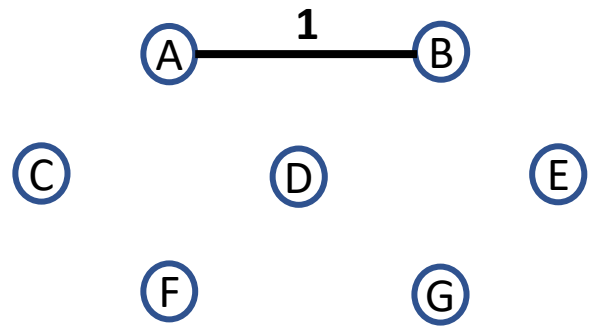
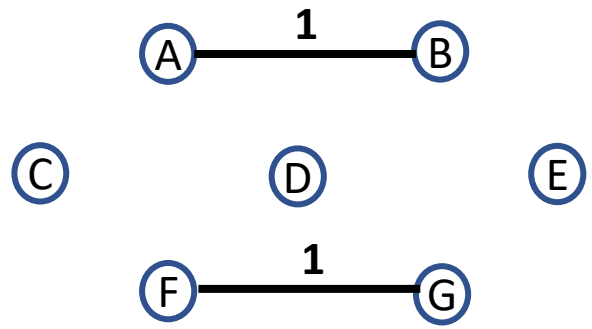
Heap (containing edges)

<A,B,1> <F,G,1> <A,D,2> <C,D,2>
<B,D,3> <A,C,4> <D,G,4><C,F,5>
<E,G,6> <D,E,7> <D,F,8> <B,E,10>

Tree Vertices

{A} {B} {C} {D} {E} {F} {G}



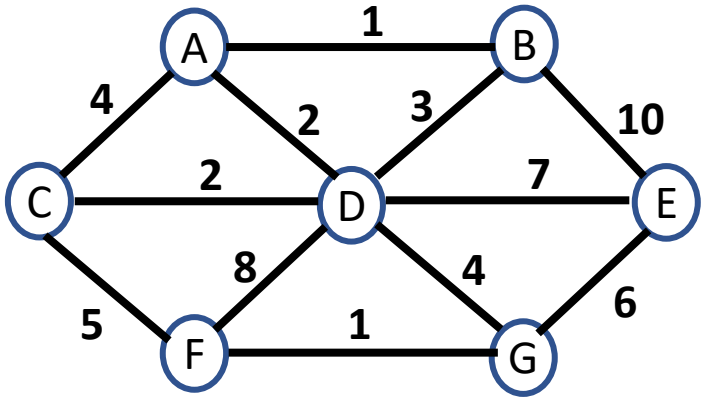
Process	Edges and Tree vertices	Spanning tree
• Delete mincost edge from heap <A,B,1> • Is edge is safe? Yes • find(A) # find(B) • Union(A,B)	**Heap (containing edges)** ~~<A,B,1>~~ <F,G,1> <A,D,2> <C,D,2> <B,D,3> <A,C,4> <D,G,4><C,F,5> <E,G,6> <D,E,7> <D,F,8> <B,E,10> **Tree Vertices** {A,B} {C} {D} {E} {F} {G} mincost = 1	
• Delete mincost edge from heap <F,G,1> • Is edge is safe? Yes • find(F) # find(G) • Union(F,G)	**Heap (containing edges)** ~~<F,G,1>~~ <A,D,2> <C,D,2> <B,D,3> <A,C,4> <D,G,4><C,F,5> <E,G,6> <D,E,7> <D,F,8> <B,E,10> **Tree Vertices** {A,B} {C} {D} {E} {F, G} mincost = 1+1=2	

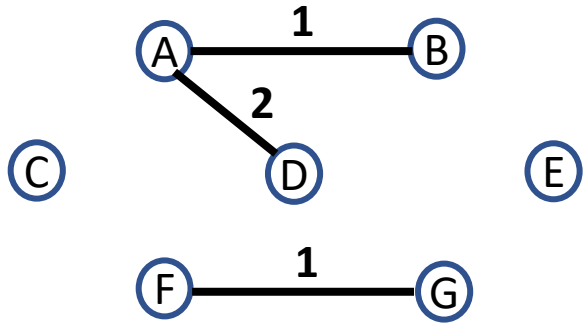
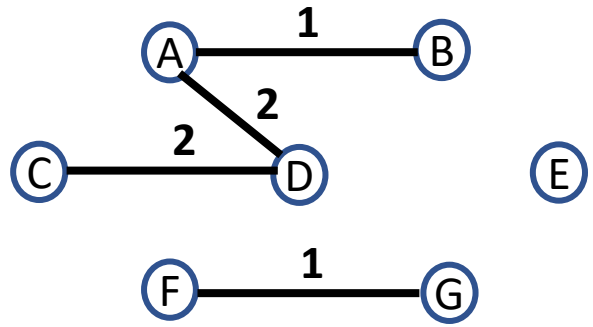
Heap (containing edges)

<A,D,2> <C,D,2> <B,D,3> <A,C,4>
<D,G,4> <C,F,5> <E,G,6> <D,E,7>
<D,F,8> <B,E,10>

Tree Vertices

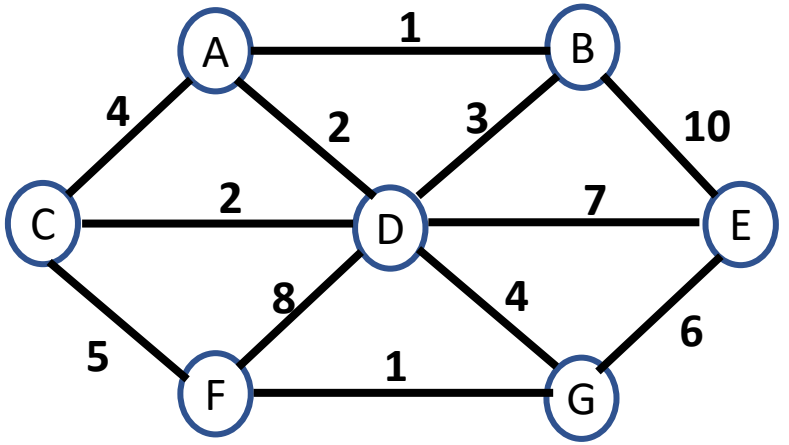
{A, B} {C} {D} {E}{F, G}

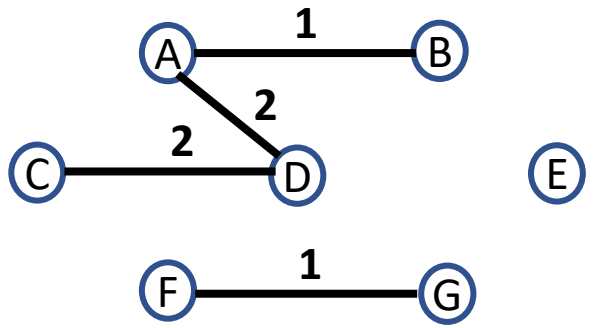
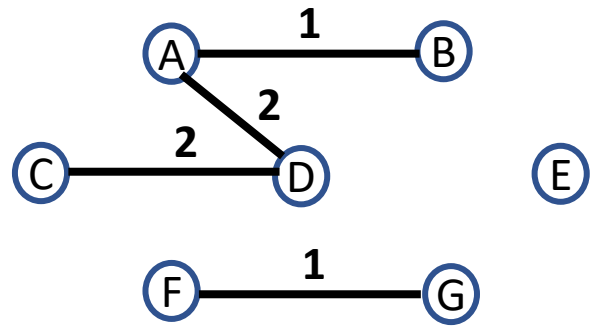


Process	Edges and Tree vertices	Spanning tree
• Delete mincost edge from heap <A,D,2> • Is edge is safe? Yes • find(A) # find(D) • Union(A,D)	**Heap (containing edges)** ~~<A,D,2>~~ <C,D,2> <B,D,3> <A,C,4> <D,G,4> <C,F,5> <E,G,6> <D, E,7> <D,F,8> <B,E,10> **Tree Vertices** {A, B, D} {C} {E} {F, G} mincost = 2+2=4	
• Delete mincost edge <u,v> from heap <C,D,2> • Is edge is safe? Yes • find(C) # find(D) • Union(C,D)	**Heap (containing edges)** ~~<C,D,2>~~ <B,D,3> <A,C,4> <D,G,4><C,F,5> <E,G,6> <D,E,7> <D,F,8> <B,E,10> **Tree Vertices** {A, B, C, D} {E} {F, G} mincost = 4+2=6	

<B,D,3> <A,C,4> <D,G,4><C,F,5>
<E,G,6> <D,E,7> <D,F,8> <B,E,10>

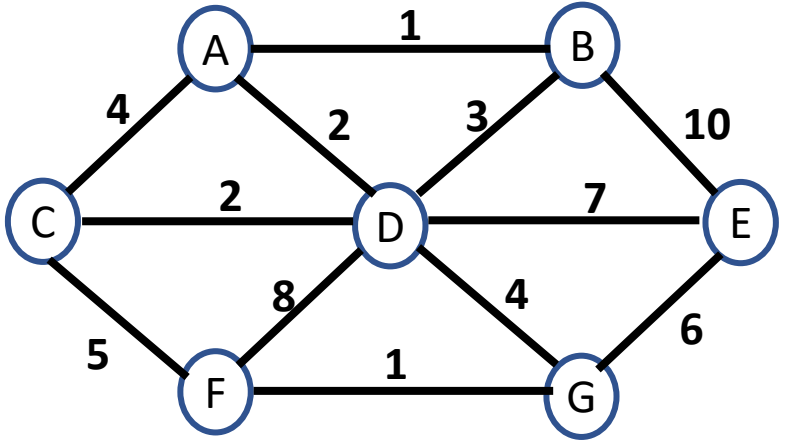
{A, B, C, D} {E} {F, G}

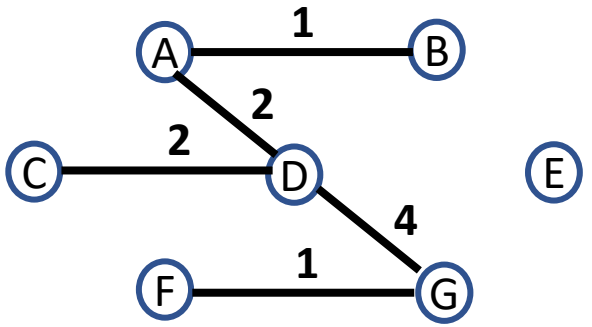
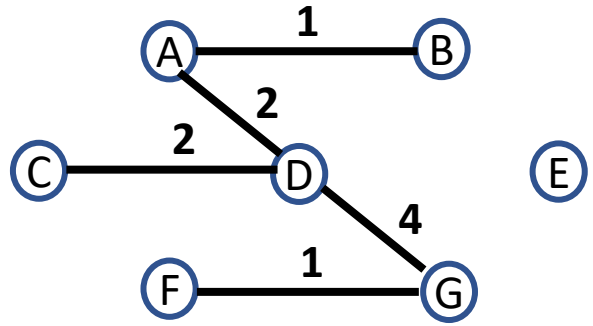


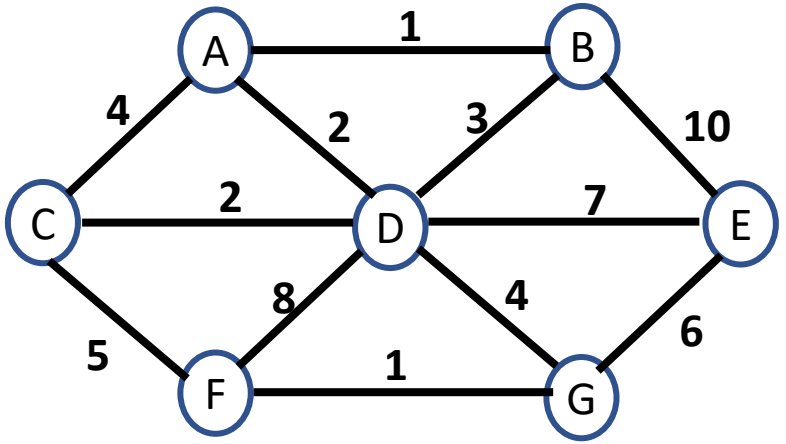
Process	Edges and Tree vertices	Spanning tree
• Delete mincost edge <u,v> from heap <B,D,3> • Is edge is safe? No • find(B) = find(D) • Discard the edge <B,D,3>	**Heap (containing edges)** ~~<B,D,3>~~ <A,C,4> <D,G,4> <C,F,5> <E,G,6> <D, E,7> <D,F,8> <B,E,10> **Tree Vertices** {A,B,C,D} {E} {F,G} mincost = 6	
• Delete mincost edge <u,v> from heap <A,C,4> • Is edge is safe? No • find(A) = find(C) • Discard the edge <A,C,4>	**Heap (containing edges)** ~~<A,C,4>~~<D,G,4><C,F,5> <E,G,6> <D,E,7> <D,F,8> <B,E,10> **Tree Vertices** {A,B,C,D} {E} {F,G} mincost = 6	

<D,G,4><C,F,5> <E,G,6> <D,E,7>
<D,F,8> <B,E,10>

{A,B,C,D} {E} {F,G}

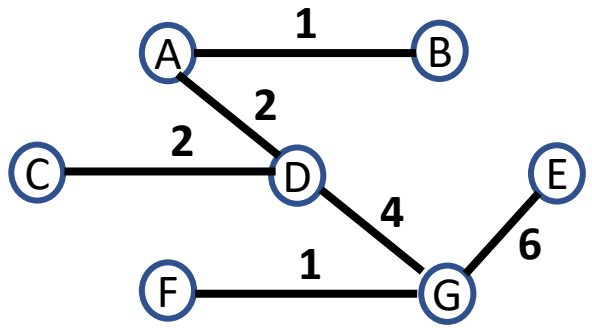


Process	Edges and Tree vertices	Spanning tree
• Delete mincost edge <u,v> from heap <D,G,4> • Is edge is safe? Yes • find(D) # find(G) • Union(D,G)	**Heap (containing edges)** ~~<D,G,4>~~ <C,F,5> <E,G,6> <D, E,7> <D,F,8> <B,E,10> **Tree Vertices** {A,B,C,D, F, G} {E} mincost = 6+4=10	
• Delete mincost edge <u,v> from heap <C,F,5> • Is edge is safe? No • find(C) = find(F) • Discard the edge <C,F,5>	**Heap (containing edges)** <C,F,5> <E,G,6> <D,E,7> <D,F,8> <B,E,10> **Tree Vertices** {A,B,C,D, F, G} {E} mincost = 10	



<E,G,6> <D,E,7> <D,F,8> <B,E,10>

{A,B,C,D, F, G} {E}

Process	Edges and Tree vertices	Spanning tree
• Delete mincost edge <u,v> from heap <E,G,6> • Is edge is safe? Yes • find(E) # find(G) • Union(E,G)	**Heap (containing edges)** ~~<D,G,4>~~ <C,F,5> <E,G,6> <D, E,7> <D,F,8> <B,E,10> **Tree Vertices** {A, B, C, D, E, F, G} mincost = 10+6=16	

Single-Source Shortest-Paths

- Given a weighted, directed graph G=(V, E) with weight function W: E->R mapping edges to real-valued weights. The weight *w(p)* of path $p = < v_0, v_1, v_2, \ldots. v_k >$ is sum of the weights of its constituent edges.

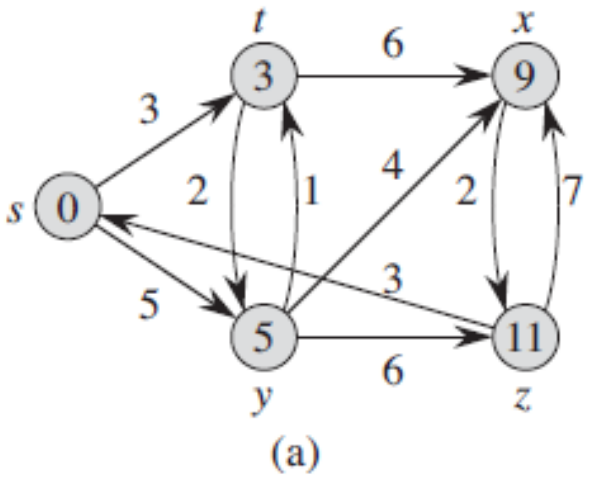
$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

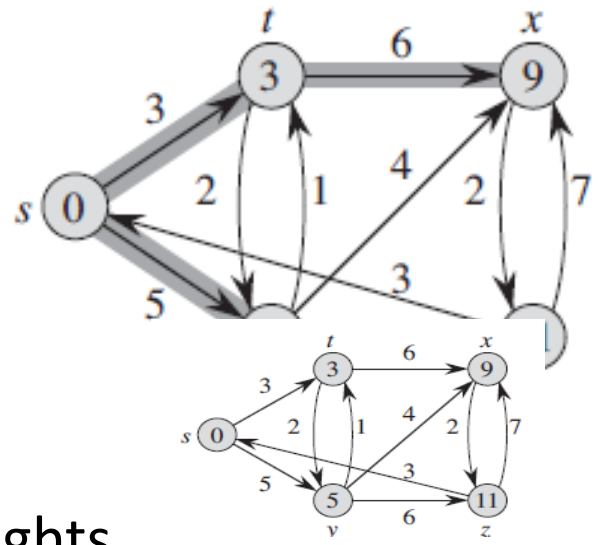
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise}. \end{cases}$$

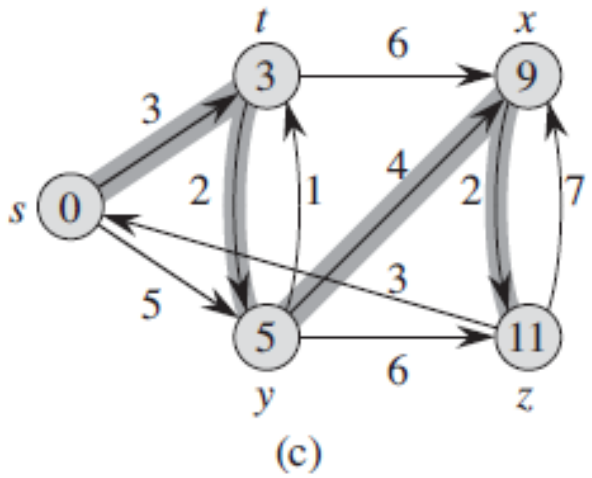
- Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that we would want to minimize.
- If the graph contains negative edges, then the shortest-paths is well defined.

Shortest-paths are not unique



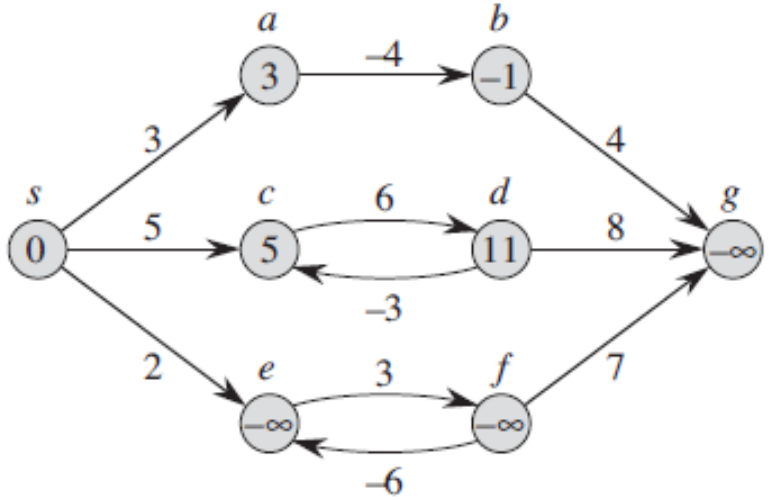
(a)

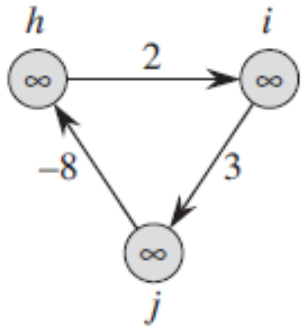




(c)

Graphs contain negative weights





Techniques used in S-P algorithms

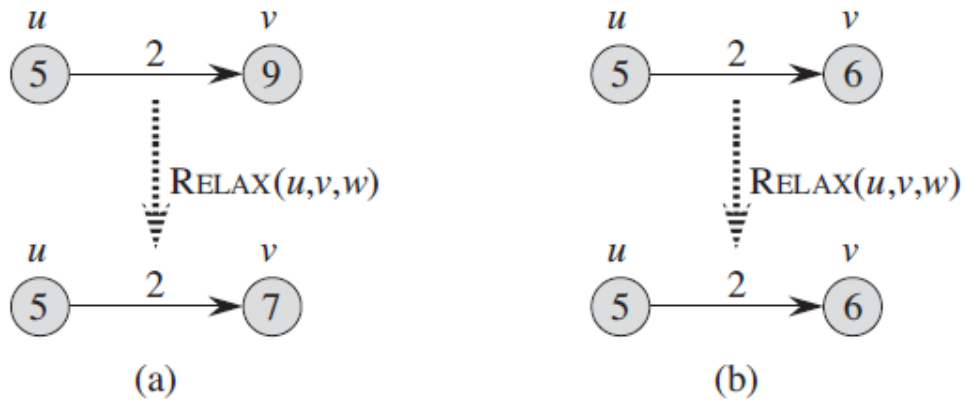
- INITIALIZE-SINGLE-SOURCE (G,s)
- RELAX (u, v, w)

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

s is the source vertex
v. *d* is shortest-path estimate from *s to v*
v. π is the predecessor of v in the shortest-path

The process of ***relaxing*** an edge *(u ,v)* consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, update *v.d* and *v.*π

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```



Single-source Shortest-Paths Algorithms

- Bellman-ford algorithm :
  - finds the shortest-paths if the graph can contain negative edges weights, but it shouldn't contain negative edge weight cycles.
  - It uses INTITALIZE-SINGLE-SOURCE(G,*s*) one time.
  - It relaxes each edge |V|-1 times

- Shortest-paths for DAG(Directed-Acyclic Graphs)

- Dijkstra's algorithm :
  - Not applicable for graphs contains negative edge weights.
  - It has lower running time than bellman ford algorithm.
  - It relaxes each edge exactly once.

Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

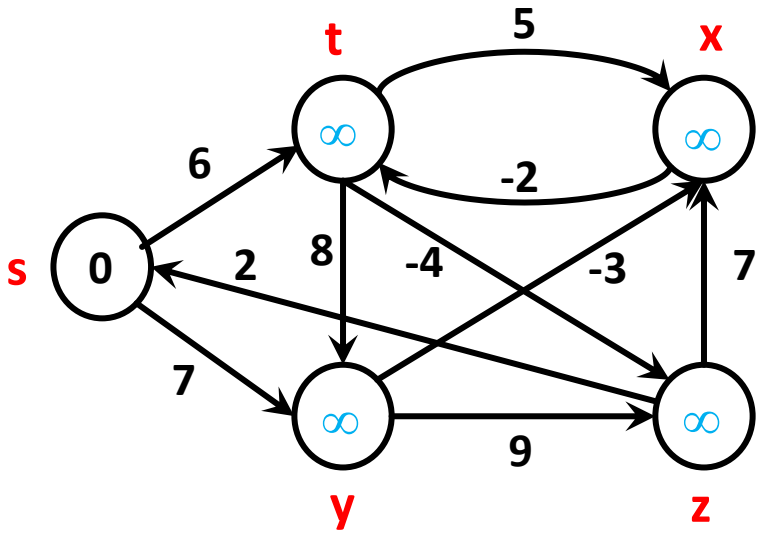
INITIALIZATION-SINGLE-SOURCE

Shortest-path estimate

	s	t	x	y	z
$v.d$	0	∞	∞	∞	∞

Predecessor graph

	s	t	x	y	z
$v.\pi$	nil	nil	nil	nil	nil

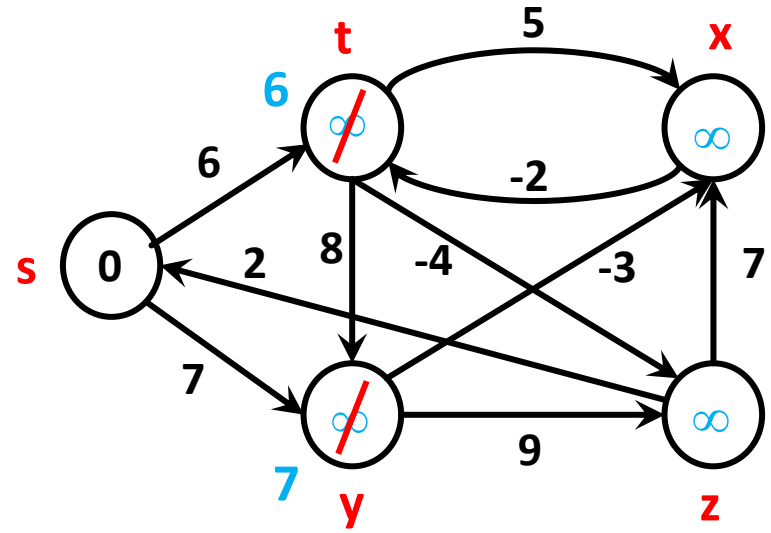


Iteration -1

		s	t	x	y	z
	$v.d$	0	~~∞~~ 6	∞	~~∞~~ 7	∞
s	0	0	6	-	7	-
t	∞	-	0	5	8	-4
x	∞	-	-2	0	-	-
y	∞	-	-	-3	0	-
z	∞	2	-	7	-	0

Predecessor graph

	s	t	x	y	z
$v.\pi$	nil	~~nil~~ s	nil	~~nil~~ s	nil

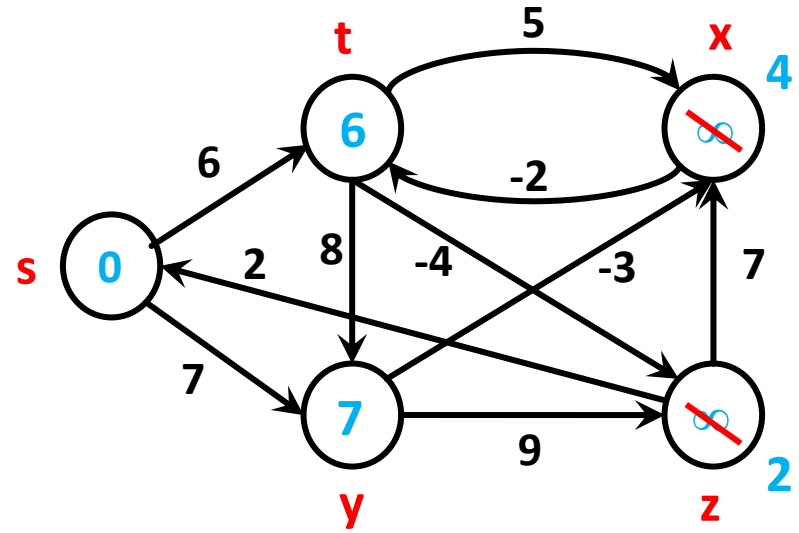


Iteration -2

		s	t	x	y	z
	$v.d$	0	6	~~∞~~ 4	7	~~∞~~ 2
s	0	0	6	-	7	-
t	6	-	0	5	8	-4
x	∞	-	-2	0	-	-
y	7	-	-	-3	0	-
z	∞	2	-	7	-	0

Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	*s*	~~*nil*~~ y	*s*	~~*nil*~~ t

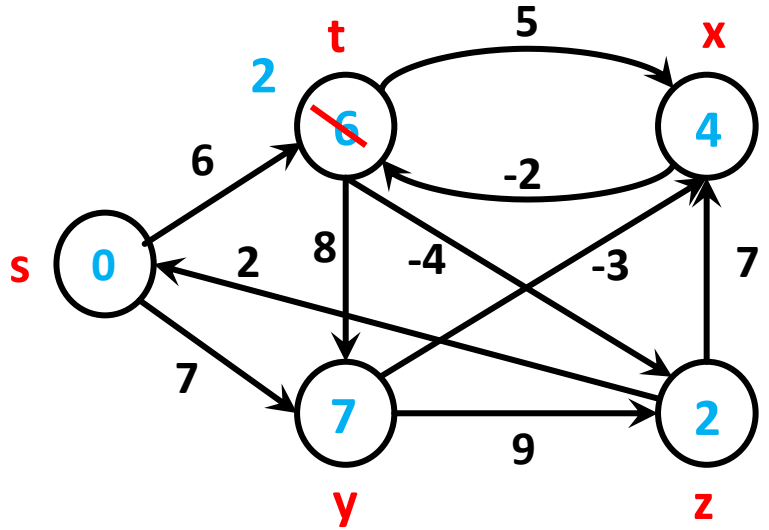


Iteration -3

		s	t	x	y	z
	$v.d$	0	~~6~~ 2	4	7	2
s	0	0	6	-	7	-
t	6	-	0	5	8	-4
x	4	-	-2	0	-	-
y	7	-	-	-3	0	-
z	2	2	-	7	-	0

Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	~~*s*~~ x	*y*	*s*	*t*

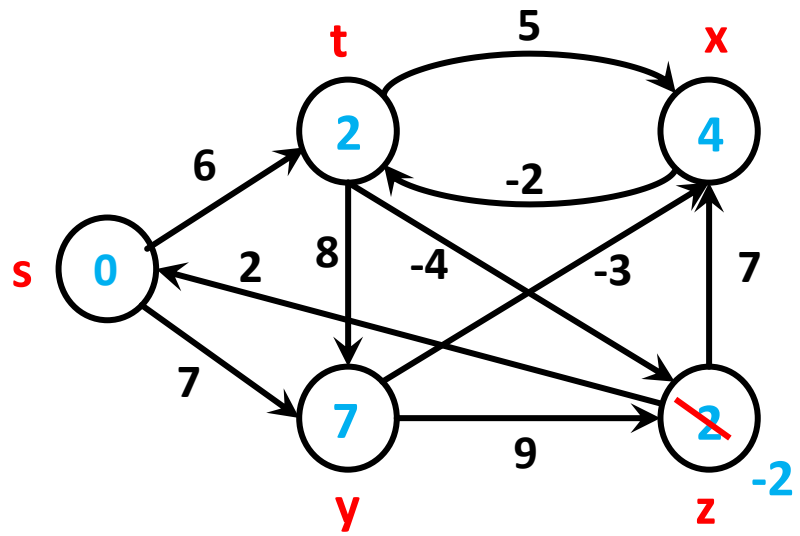


Iteration -4

		s	t	x	y	z
	$v.d$	0	2	4	7	~~2~~ -2
s	0	0	6	-	7	-
t	2	-	0	5	8	-4
x	4	-	-2	0	-	-
y	7	-	-	-3	0	-
z	2	2	-	7	-	0

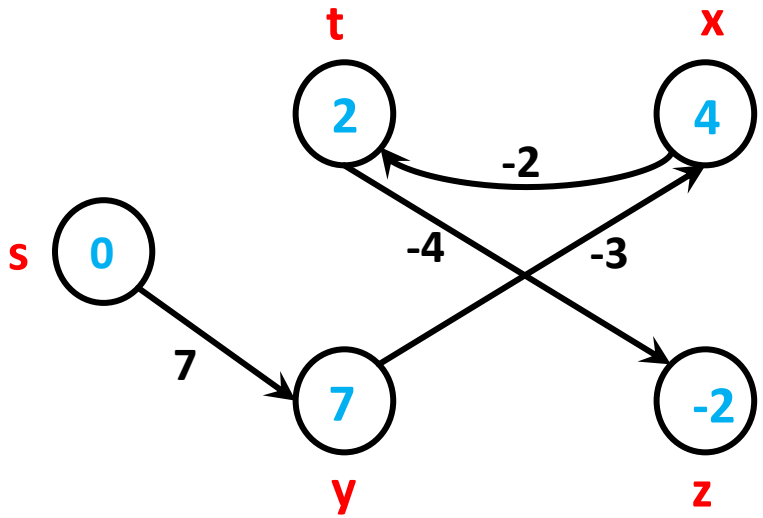
Predecessor graph

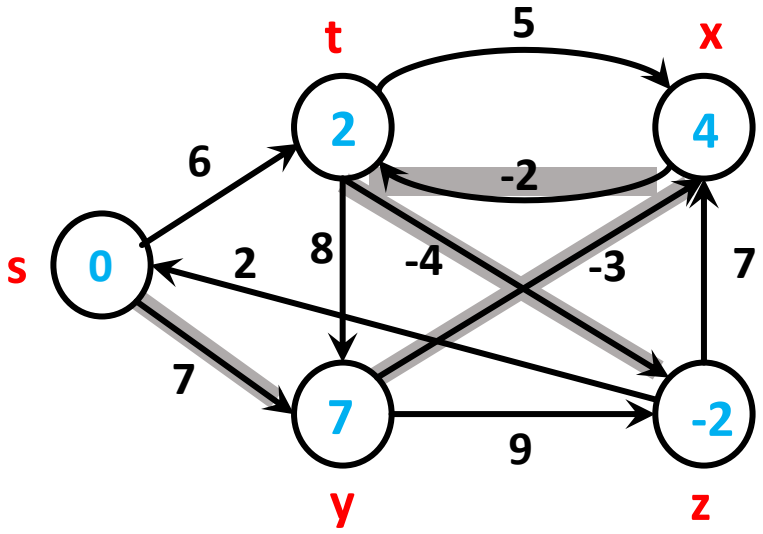
	s	t	x	y	z
$v.\pi$	*nil*	x	y	s	t



Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	x	y	s	t





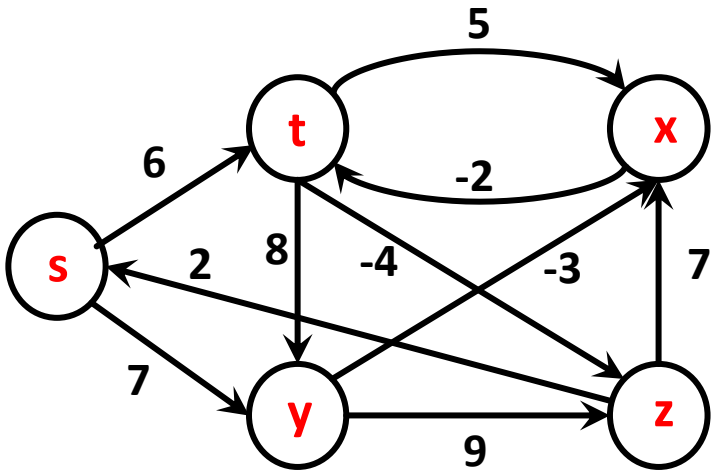
Predecessor graph

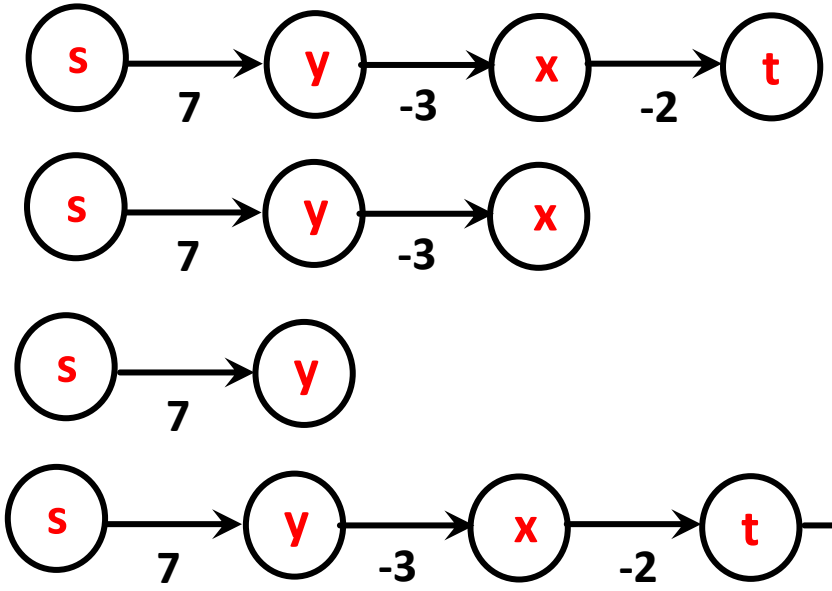
Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	*x*	*y*	*s*	*t*

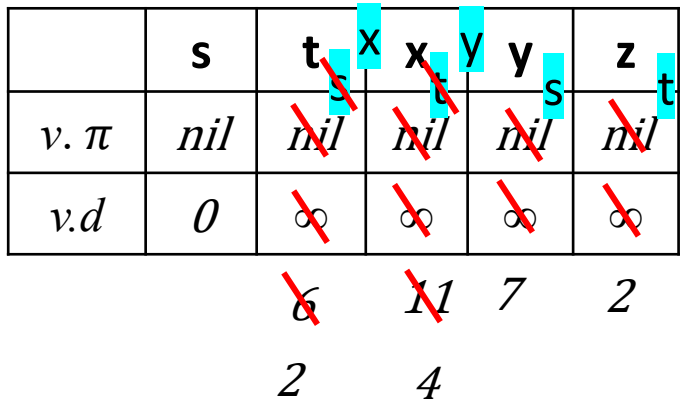
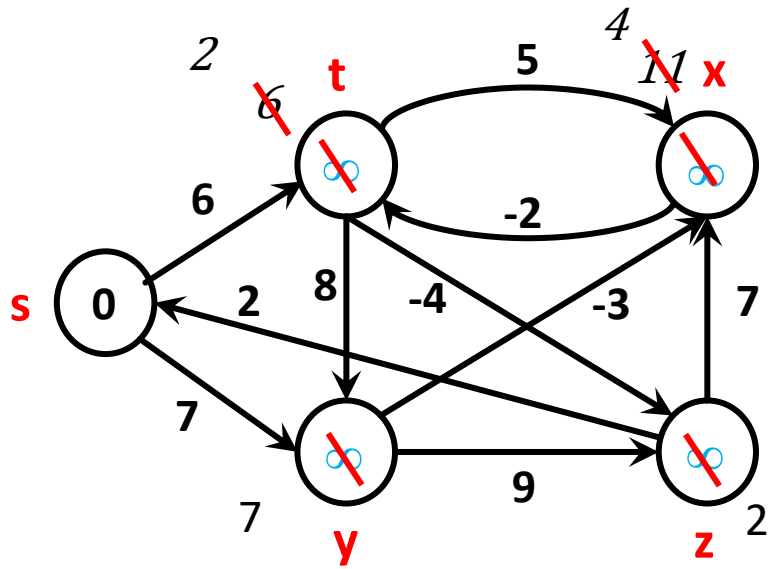
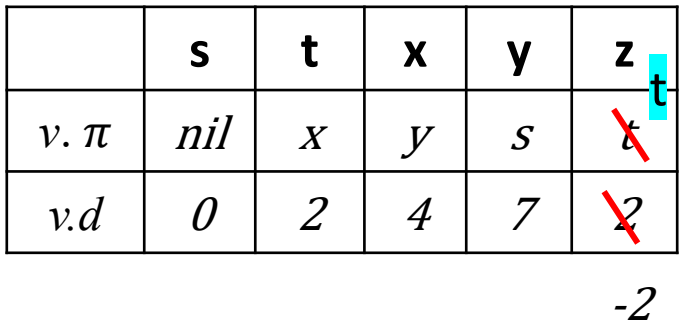
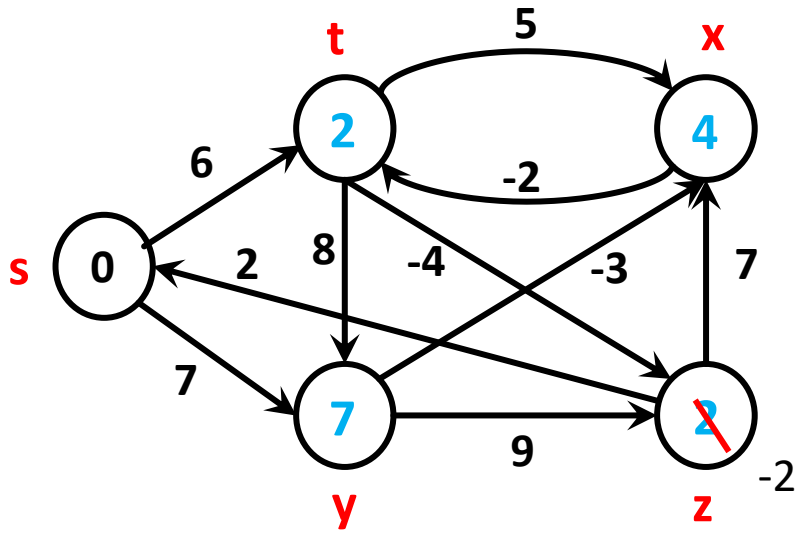
Shortest-path distance from s

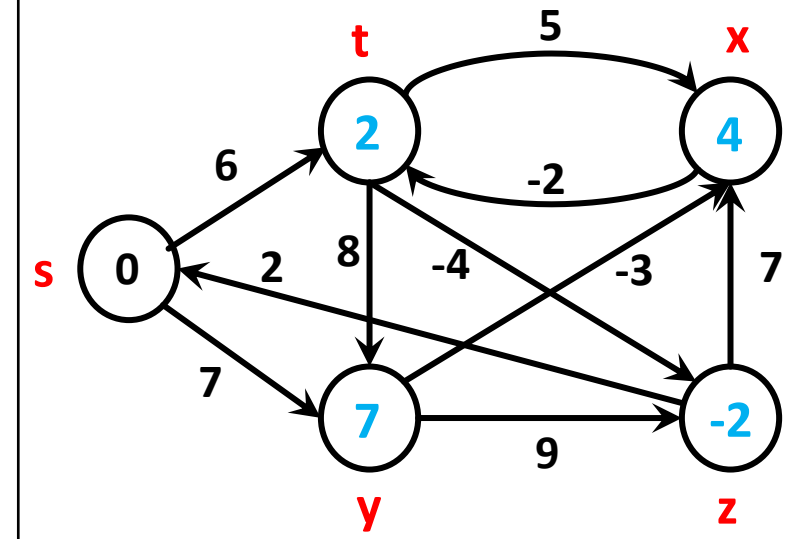
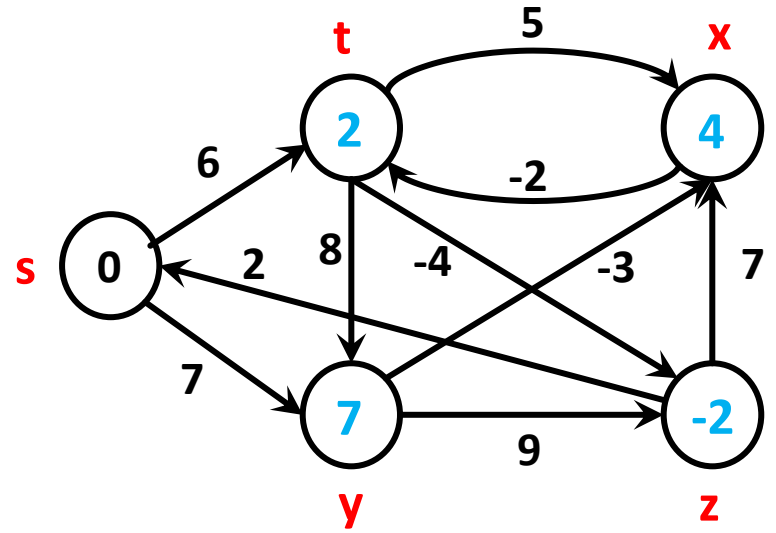
	s	t	x	y	z
v.d	0	2	4	7	-2



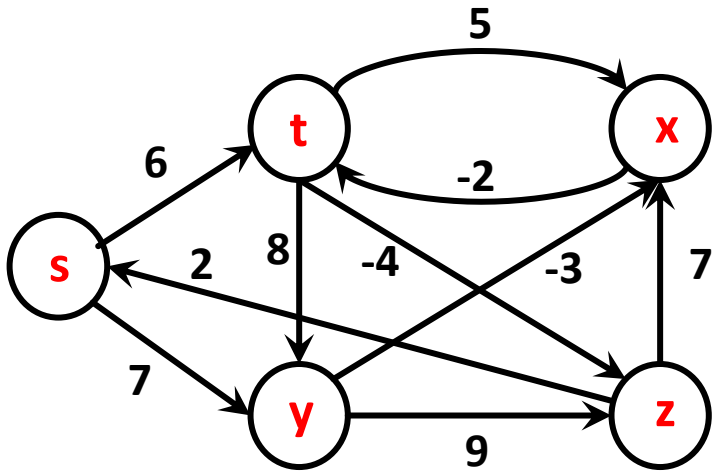



```
print_path(G,s,v)
1. If v==s
2.       print s
3. else if v.pi =nil
4.        return
5. else
6.       print_path(G,s,v.pi)
7. print v
```

Iteration	Process	Data structures status	Shortest-path instance
1	Relax-each edge <s , t> <y , z> <s , y> <y , x> <t , x> <x , t> <t , y> <z , x> <t , z> <z , s>		
2	Relax-each edge <s , t> <y , x> <s , y> <y , z> <t , x> <x , t> <t , y> <z , x> <t , z> <z , s>		

<table>
<tr><th>Iteration</th><th>Process</th><th>Data structures used</th><th>Shortest-path instance</th></tr>
<tr>
<td>3</td>
<td>Relax-each edge
<s , t> <y , z>
<s , y> <t , z>
<t , x> <x , t>
<t , y> <z , x>
<y , x> <z , s></td>
<td>
<table>
<tr><th></th><th>s</th><th>t</th><th>x</th><th>y</th><th>z</th></tr>
<tr><td>v. π</td><td>nil</td><td>x</td><td>y</td><td>s</td><td>t</td></tr>
<tr><td>v.d</td><td>0</td><td>2</td><td>4</td><td>7</td><td>-2</td></tr>
</table>
</td>
<td> 
</td>
</tr>
<tr>
<td>4</td>
<td>Relax-each edge
<s , t> <y , z>
<s , y> <t , z>
<t , x> <x , t>
<t , y> <z , x>
<y , x> <z , s></td>
<td>
<table>
<tr><th></th><th>s</th><th>t</th><th>x</th><th>y</th><th>z</th></tr>
<tr><td>v. π</td><td>nil</td><td>x</td><td>y</td><td>s</td><td>t</td></tr>
<tr><td>v.d</td><td>0</td><td>2</td><td>4</td><td>7</td><td>-2</td></tr>
</table>
</td>
<td> 
</td>
</tr>
</table>

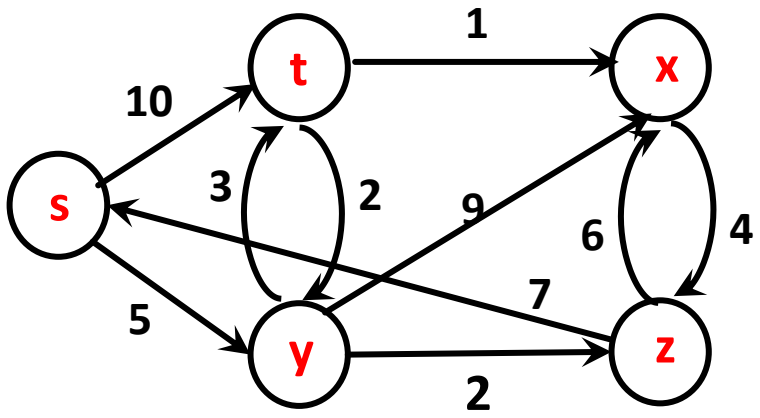
0



Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G =(V, E) for the case in which all edge weights are nonnegative.
- It selects the vertex $u \in V - s$ with the minimum shortest-path estimate and adds to S.
- Relaxes all the edges ,which are leaving from u.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```



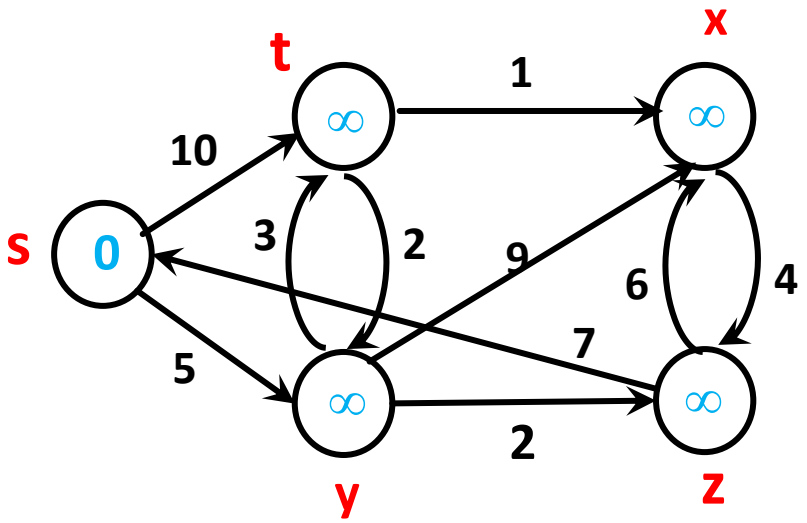
INITIALIZATION-SINGLE-SOURCE

Shortest-path estimate

	s	t	x	y	z
$v.d$	0	∞	∞	∞	∞

Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	*nil*	*nil*	*nil*	*nil*

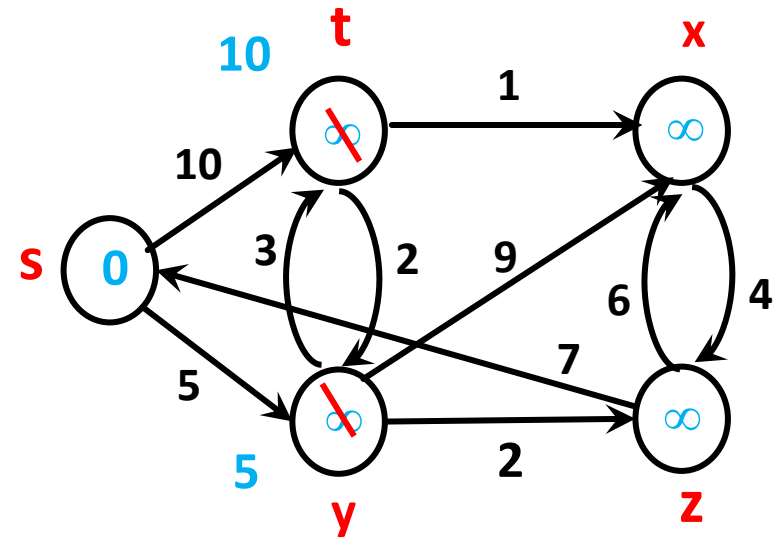


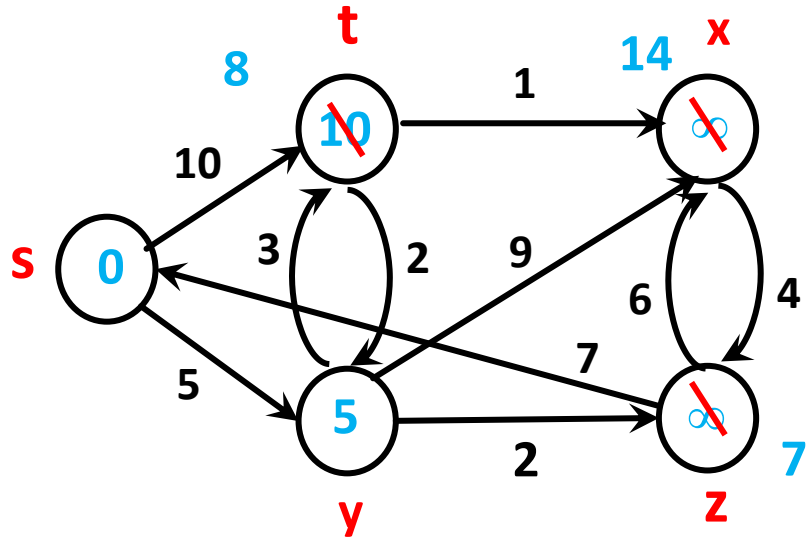
Predecessor graph

	s	t	x	y	z
v. π	*nil*	~~*s*~~ y	~~*nil*~~ y	*s*	~~*nil*~~ y

			v. *d*				
S	**Vertex Selected**	**Relax**	**s**	**t**	**x**	**y**	**z**
	s	(s,t) and (s,y)	*0*	~~*10*~~ ∞	∞	~~*5*~~ ∞	∞

{s}	**y**	**(y, t), (y,x) and (y,z)**	0	*8*	*14*	*5*	*7*

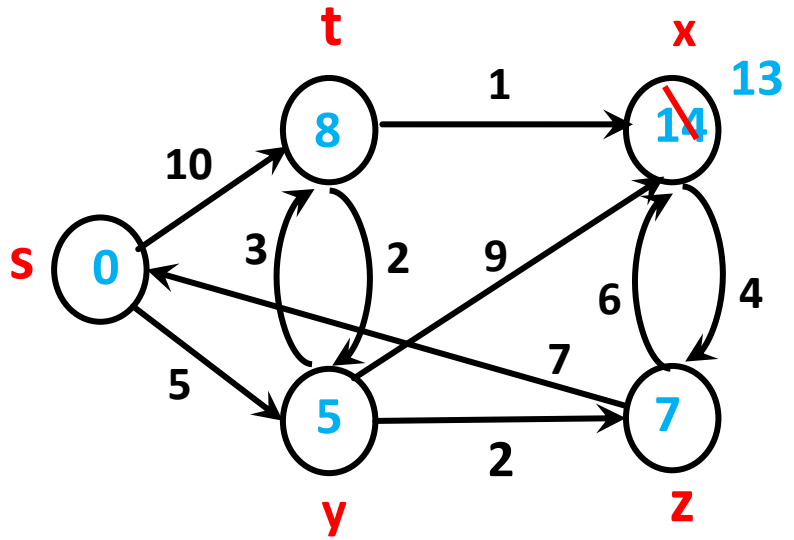


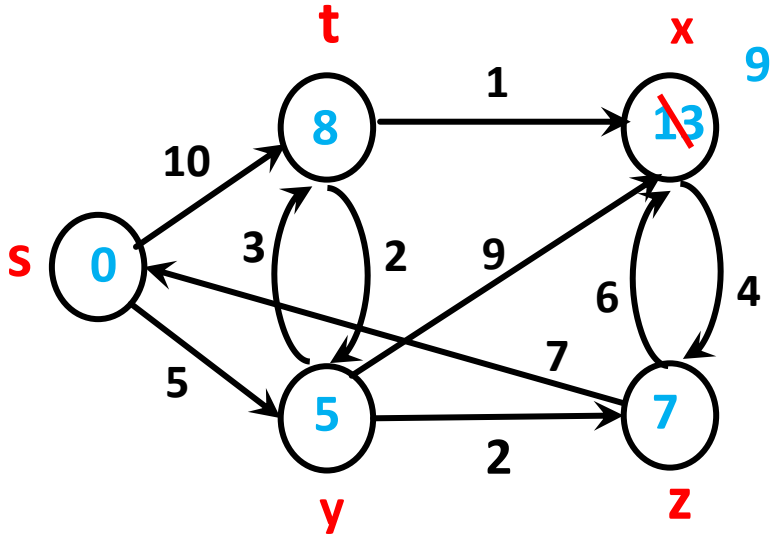


Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	*y*	*nil*	*s*	*y*

			$v.d$				
S	**Vertex Selected**	**Relax**	**s**	**t**	**x**	**y**	**z**
{s}	**s**	(s,t) and (s,y)	*0*	*10*	∞	*5*	∞
{s}	**y**	**(y, t), (y,x) and (y,z)**	*0*	*8*	*14*	*5*	*7*
{s,y}	**z**	**(z, s), (z,x)**	*0*	*8*	*13*	*5*	*7*
{s,y,z}	**t**	**(t, y), (t,x)**	*0*	*8*	*9*	*5*	*7*

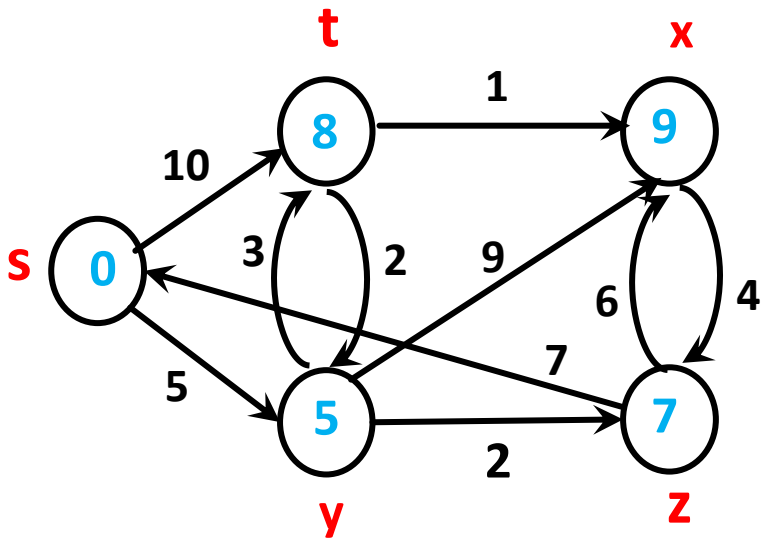




Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	*y*	~~*nil*~~ t	*s*	*y*

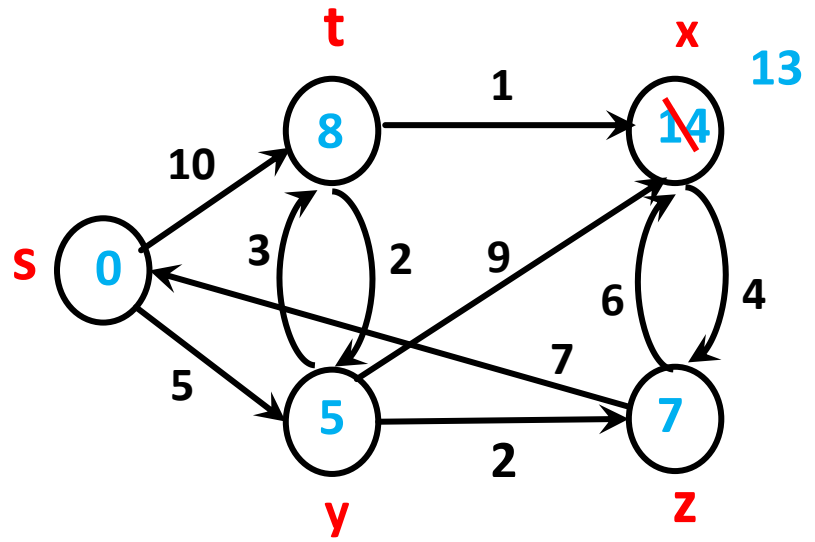
			$v.d$				
S	**Vertex Selected**	**Relax**	**s**	**t**	**x**	**y**	**z**
{s}	**s**	(s,t) and (s,y)	*0*	*10*	∞	*5*	∞
{s,y}	**y**	**(y, t), (y,x) and (y,z)**	*0*	*8*	*14*	*5*	*7*
{s,y,z}	**z**	**(z, s), (z,x)**	*0*	*8*	*13*	*5*	*7*
{s, y, z, t}	**t**	**(t, y), (t,x)**	*0*	*8*	*9*	*5*	*7*
{s, y, z, t, x}	**x**	**(x, z)**	*0*	*8*	*9*	*5*	*7*

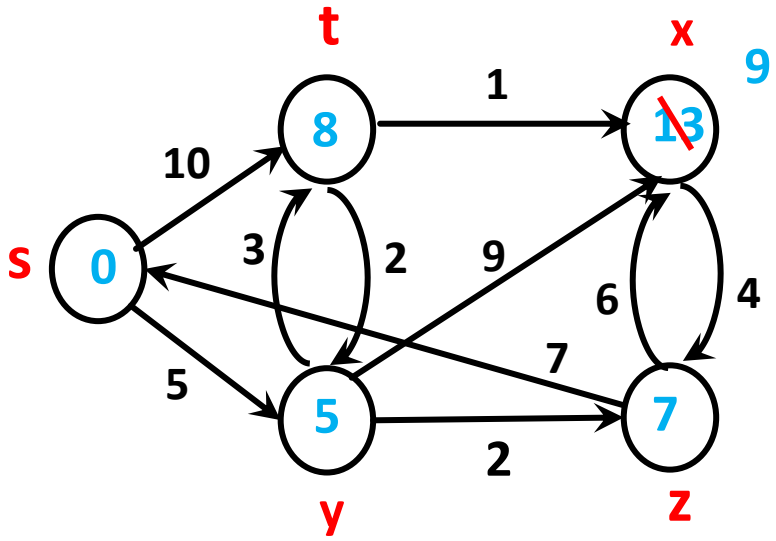


Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	*y*	~~*nil*~~ t	*s*	*y*

			$v.d$				
S	**Vertex Selected**	**Relax**	**s**	**t**	**x**	**y**	**z**
{s}	**s**	(s,t) and (s,y)	*0*	~~∞~~ *10*	∞	~~∞~~ *5*	∞
{s}	**y**	**(y, t), (y,x) and (y,z)**	*0*	~~*10*~~ *8*	~~∞~~ *14*	*5*	~~∞~~ *7*
{s,y}	**z**	**(z, s), (z,x)**	*0*	*8*	~~*14*~~ *13*	*5*	*7*
{s,y,z}	**t**	**(t, y), (t,x)**	*0*	*8*	~~*13*~~ *9*	*5*	*7*
{s,y,z,t}	**x**	**(x,z)**	*0*	*8*	*9*	*5*	*7*

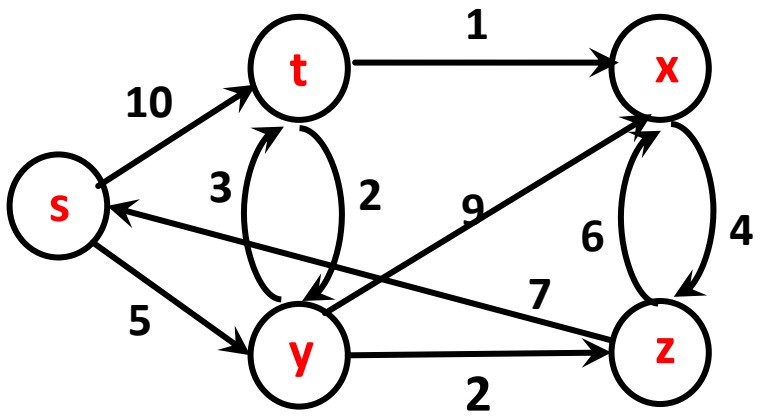




Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G =(V, E) for the case in which all edge weights are nonnegative.
- It selects the vertex $u \in V - s$ with the minimum shortest-path estimate and adds to S.
- Relaxes all the edges ,which are leaving from u.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```



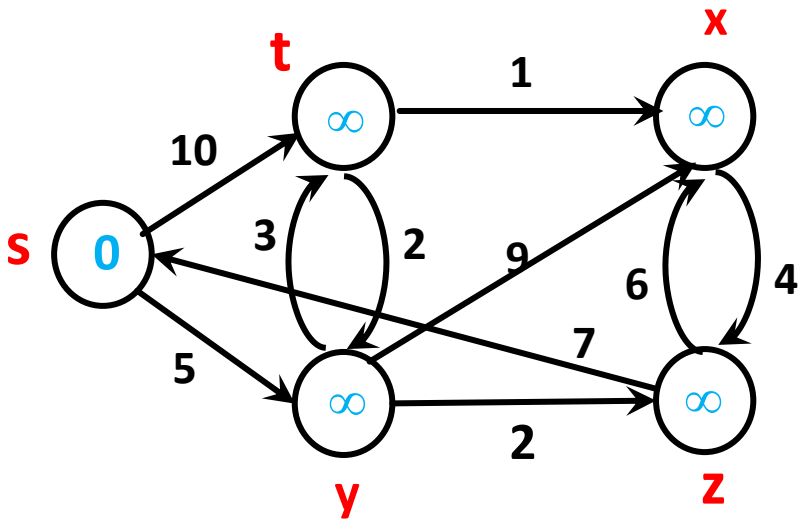
INITIALIZATION-SINGLE-SOURCE

Shortest-path estimate

	s	t	x	y	z
$v.d$	0	∞	∞	∞	∞

Predecessor graph

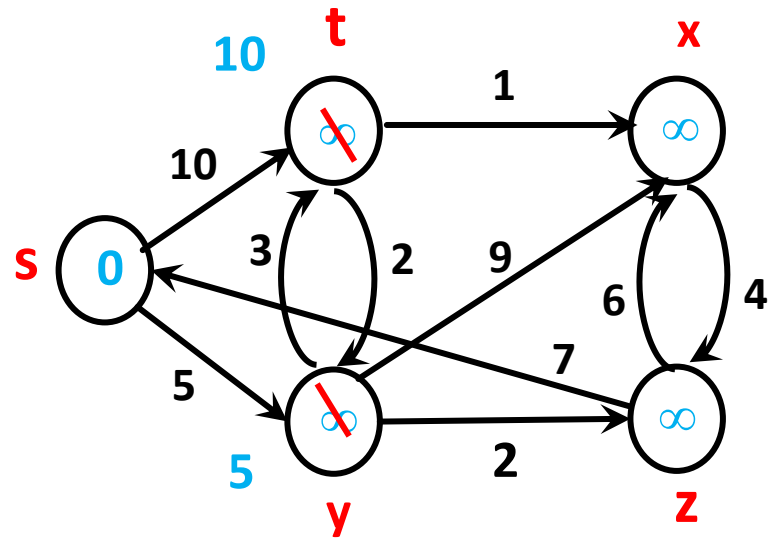
	s	t	x	y	z
$v.\pi$	*nil*	*nil*	*nil*	*nil*	*nil*

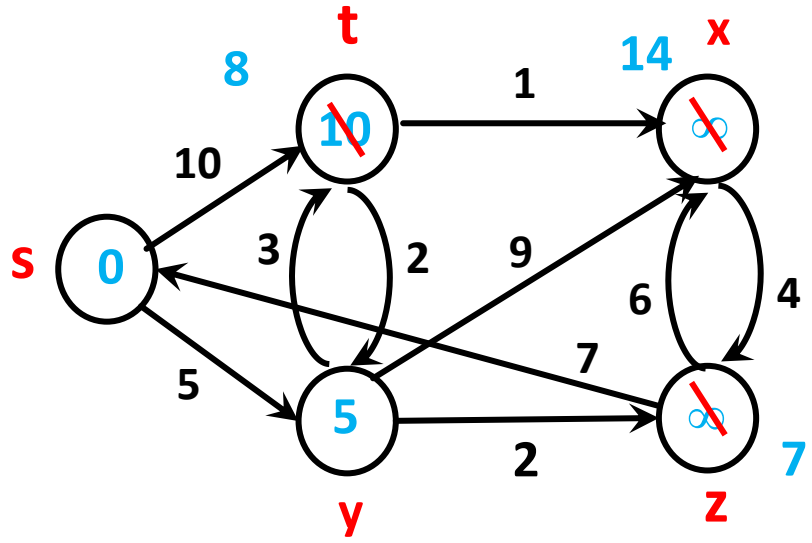


Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	*nil*	*nil*	*nil*	*nil*

S (shortest-path vertices)	Q (Remaining vertices)	Vertex Selected	Relax	$v.d$				
				s	t	x	y	z
	{s,t,x,y,z}			*0*	∞	∞	∞	∞
{s}	{t,x,y,z}	s	(s,t) and (s,y)	0	*10*	∞	*5*	∞
{s,y }	{t,x,z}	y	(y, t), (y,x) and (y,z)	0	*8*	*14*	*5*	*7*

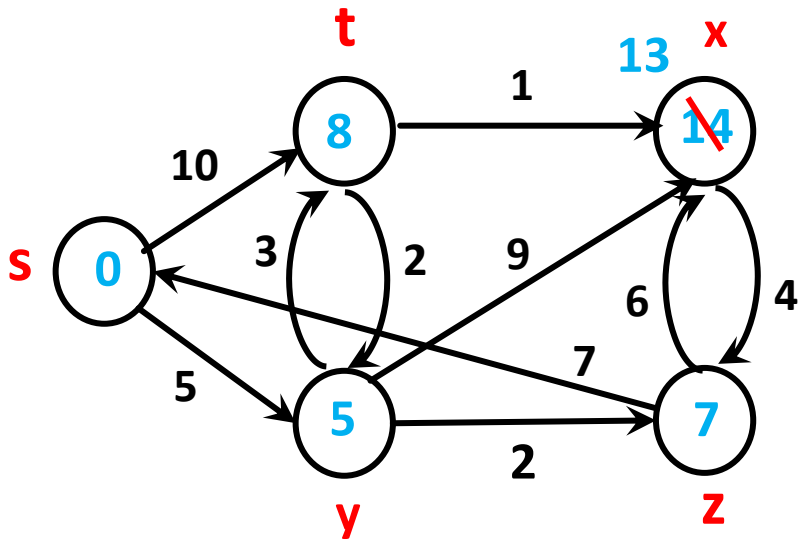


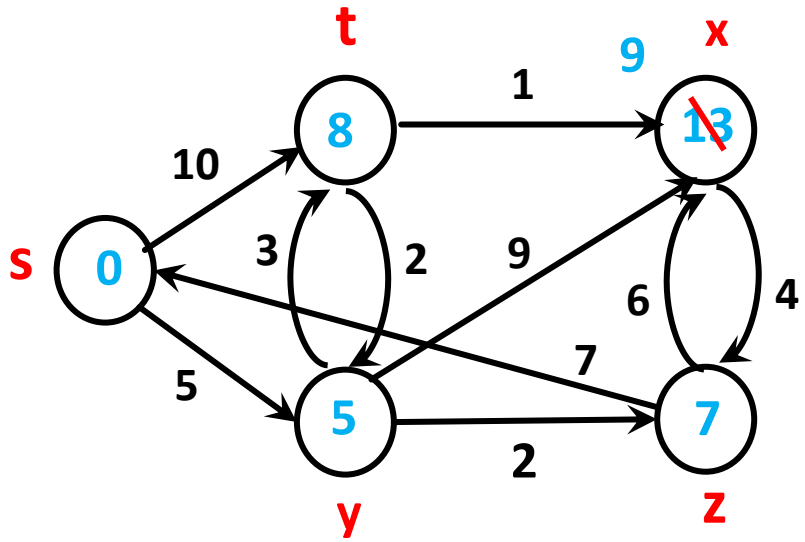


Predecessor graph

	s	t	x	t	y	z
			z			
$v.\pi$	*nil*	*y*	~~y~~		*s*	*y*

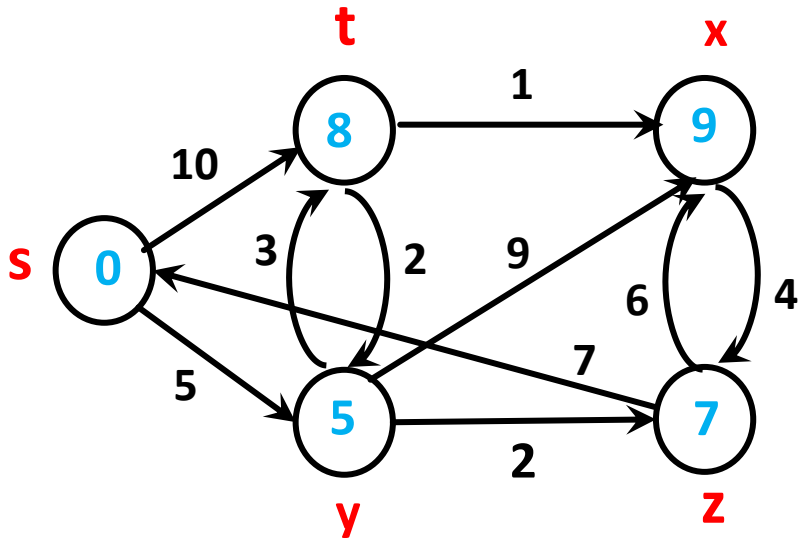
S (shortest-path vertices)	Q (Remaining vertices)	Vertex Selected	Relax	$v.d$				
				s	t	x	y	z
	{s,t,x,y,z}			*0*	∞	∞	∞	∞
{s}	{t,x,y,z}	s	(s,t) and (s,y)	*0*	10	∞	5	∞
{s,y}	{t,x,z}	y	(y,t), (y,x) and (y,z)	*0*	8	14	5	7
{s,y,z }	{t,x}	z	(z, s), (z,x)	0	*8*	*13*	*5*	*7*
{s,y,z,t }	{x}	t	(t, x), (t,y)	0	*8*	*9*	*5*	*7*



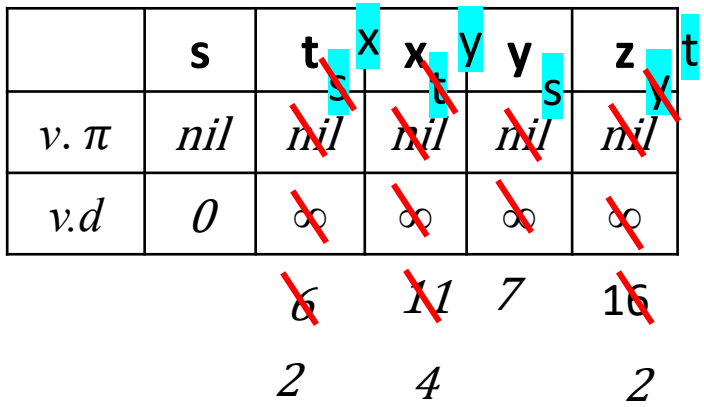
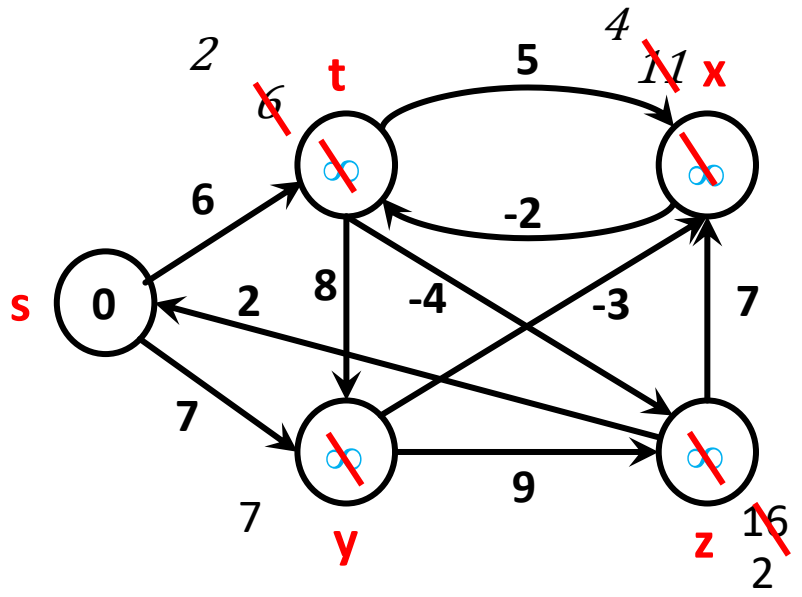
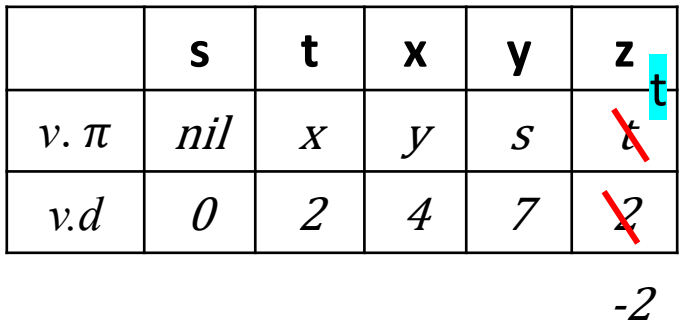
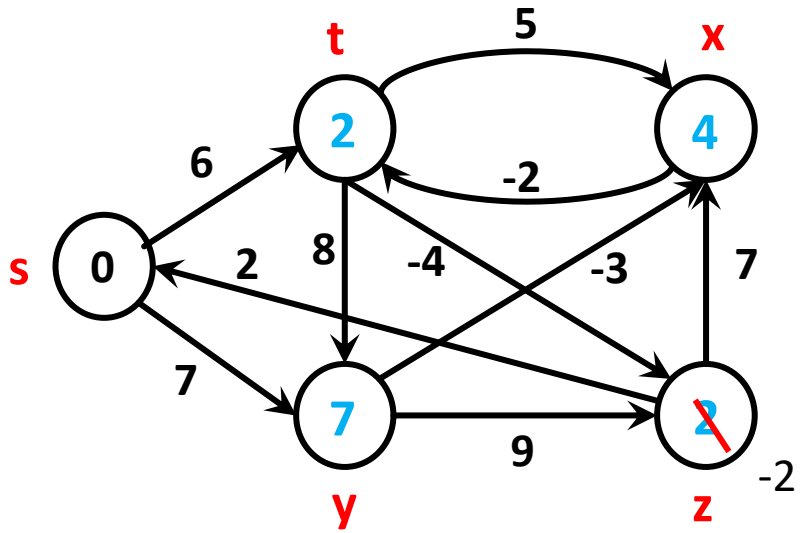


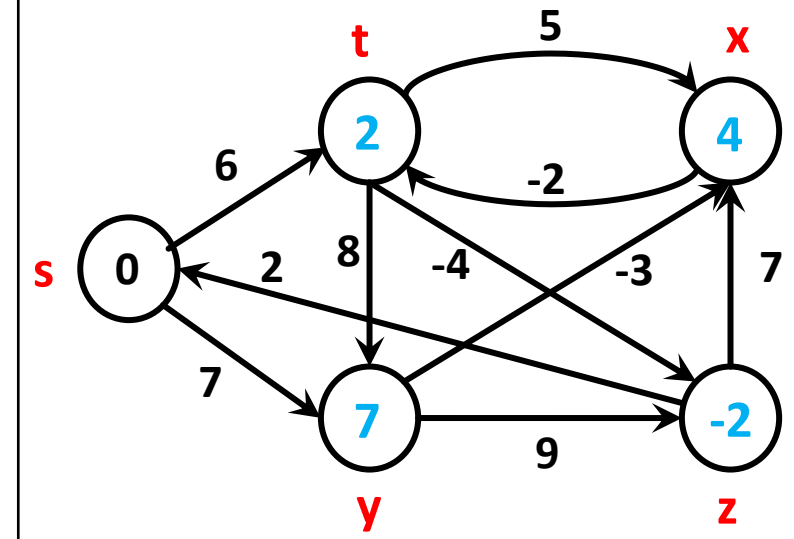
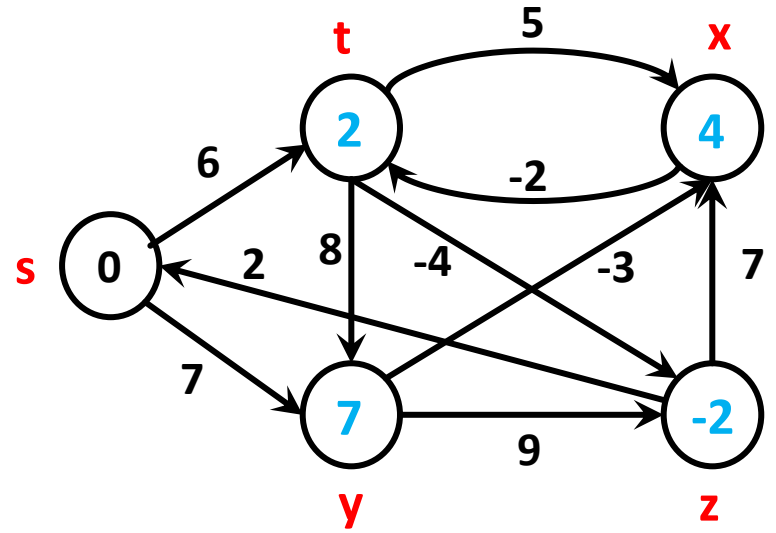
Predecessor graph

	s	t	x	y	z
$v.\pi$	*nil*	y	t	s	y



S (shortest-path vertices)	Q (Remaining vertices)	Vertex Selected	Relax	$v.d$				
				s	t	x	y	z
	{s,t,x,y,z}			0	∞	∞	∞	∞
{s}	{t,x,y,z}	s	(s,t) and (s,y)	0	10	∞	5	∞
{s,y}	{t,x,z}	y	(y,t), (y,x) and (y,z)	0	8	14	5	7
{s,y,z}	{t,x}	z	(z,s), (z,x)	0	8	13	5	7
{s,y,z,t}	{x}	t	(t,x),(t,y)	0	8	9	5	7
{s,y,z,t,x}	{}	x	(x,z)	0	8	9	5	7

Iteration	Process	Data structures status	Shortest-path instance
1	Relax-each edge <s , t> <y , z> <s , y> <t , z> <t , x> <x , t> <t , y> <z , x> <y , x> <z , s>		
2	Relax-each edge <s , t> <y , x> <s , y> <y , z> <t , x> <x , t> <t , y> <z , x> <t , z> <z , s>		

<table>
<tr><th>Iteration</th><th>Process</th><th>Data structures used</th><th>Shortest-path instance</th></tr>
<tr>
<td>3</td>
<td>Relax-each edge
<s , t> <y , z>
<s , y> <t , z>
<t , x> <x , t>
<t , y> <z , x>
<y , x> <z , s></td>
<td>
<table>
<tr><th></th><th>s</th><th>t</th><th>x</th><th>y</th><th>z</th></tr>
<tr><td>v. π</td><td>nil</td><td>x</td><td>y</td><td>s</td><td>t</td></tr>
<tr><td>v.d</td><td>0</td><td>2</td><td>4</td><td>7</td><td>-2</td></tr>
</table>
</td>
<td> 
</td>
</tr>
<tr>
<td>4</td>
<td>Relax-each edge
<s , t> <y , z>
<s , y> <t , z>
<t , x> <x , t>
<t , y> <z , x>
<y , x> <z , s></td>
<td>
<table>
<tr><th></th><th>s</th><th>t</th><th>x</th><th>y</th><th>z</th></tr>
<tr><td>v. π</td><td>nil</td><td>x</td><td>y</td><td>s</td><td>t</td></tr>
<tr><td>v.d</td><td>0</td><td>2</td><td>4</td><td>7</td><td>-2</td></tr>
</table>
</td>
<td> 
</td>
</tr>
</table>