UNIT – IV

Greedy Algorithms - Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure.

Elements of the greedy strategy-

More generally, we design greedy algorithms according to the following sequence
of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes
   the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Greedy-choice property –

we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems.

Optimal substructure

A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. if an optimal solution to subproblem Sij includes an activity ak, then it must also contain optimal solutions to the subproblems Sik and Skj. Given this optimal substructure, we argued that if we knew which activity to use as ak, we could construct an optimal solution to Sij by selecting ak along with all activities in optimal solutions to the subproblems Sik and Skj . optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem***-because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy.

Knapsack Problem –

Now, let us try to apply the greedy method to solve a more complex problem. This problem is the knapsack problem. We are given n objects and a knapsack. Object i has a weight w; 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 of PiXi 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 may be stated as:

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

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

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

The profits and weights are positive numbers.

Fractional Knapsack, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem. An **efficient solution** is to use Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

Algorithm 4.3 Algorithm for greedy strategies for the knapsack problem

```
procedure 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 +1) / W(i + 1). M is the knapsack size and
//X(l :n) is the solution vector

real P(l:n), W(l:n), X(l:n), M. cu;
integer i, n ;
X ← 0  //initialize solution to zero
cu ← M  // cu = remaining knapsack capacity
for i - 1 ton do
if W(i) > cu then exit endif
X(i) ← 1
cu ← cu - W(i)
repeat
    if i <= n then X(i) ← cu/W(i) endif
    end GREEDY _KNAPSACK
```

Analysis

If the provided items are already sorted into a decreasing order of P_i / W_i, then the it takes a time in O(n), Therefore, the total time including the sort is in O(n log n).

Q. $w = 21$

$P(10, 5, 15, 7, 6, 18, 3)$

$w(2, 3, 5, 7, 1, 4, 1)$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
P	10	5	15	7	6	18	3
W	2	3	5	7	1	4	1
P/W	5	1.67	3	1	6	4.5	3

	x_5	x_1	x_6	x_7	x_3	x_2	x_4
P	6	10	18	3	15	5	7
W	1	2	4	1	5	3	7
P/W	6	5	4.5	3	3	1.67	1

object	individual weight	total weight	object profit.	total profit	condition
x_5	1	$0+1=1$	6	$0+6=6$	$1 \leq 21$ T
x_1	2	$1+2=3$	10	$6+10=16$	$3 \leq 21$ T
x_6	4	$3+4=7$	18	$16+18=34$	$7 \leq 21$ T
x_7	1	$7+1=8$	3	$34+3=37$	$8 \leq 21$ T
x_3	5	$8+5=13$	15	$37+15=52$	$13 \leq 21$ T
x_2	3	$13+3=16$	5	$52+5=57$	$16 \leq 21$ T
x_4	7	$16+5=21$	7	$57+5=62$	$21 \leq 21$ T

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	1	1	0.71	1	1	1

Q1 solve the knapsack problem for given data

⇒ n = 5

W = 100

w(10, 20, 30, 40, 50) P(20, 50, 66, 40, 60)

	x_1	x_2	x_3	x_4	x_5
• P	20	30	66	40	60
• W	10	20	30	40	50
• P	2	2	11	1	6
W		3	5		5
• P/W	2	1.5	2.2	1	1.2

	x_3	x_1	x_2	x_5	x_4
P	66	20	30	60	40
W	30	10	20	50	40
P/W	2.2	2	1.5	1.2	1

object	profit	W remaining
x_3	66	100 - 30 = 70
x_1	20	70 - 10 = 60
x_2	30	60 - 20 = 40
x_5	40×1.2 = 48	40 - 40 = 0
	164	

x_1	x_2	x_3	x_4	x_5
1	1	1	0	0.8

Q.2 n = 4, W = 75

P(280, 100, 120, 120)

w(40, 10, 20, 24)

⇒

	x_1	x_2	x_3	x_4
P	280	100	120	120
W	40	10	20	24
P/W	7	10	6	5

	x_2	x_1	x_3	x_4
P	100	280	120	120
W	10	40	20	24
P/W	10	7	6	5

Object	profit	W remaining.
x_2	100	0 + 10 = 10
x_1	280	10 + 40 = 50
x_3	120	20 + 50 = 70
x_4	5×5 = 25	70 + 5 = 75
	525	

x_1	x_2	x_3	x_4
1	1	1	0.20854

24.2-4
Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm.

Single source shortest path algorithm using Greedy Approach

24.3 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. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```

Dijkstra's algorithm relaxes edges as shown in Figure 24.6. Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from $Q = V - S$ and line 6 adds it to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u. Observe that the algorithm never inserts vertices into Q after line 3 and that each vertex is extracted from Q

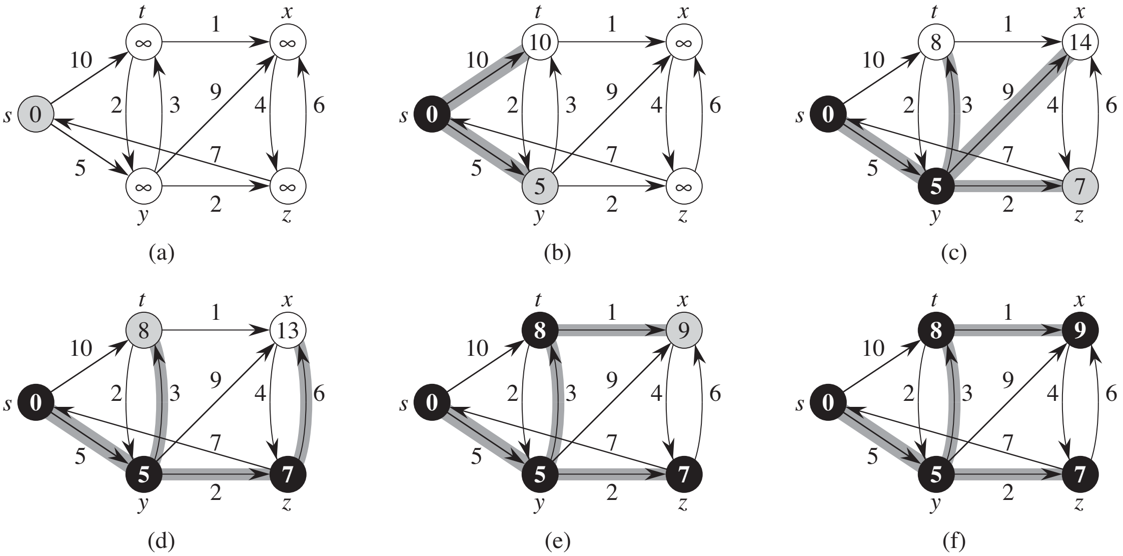


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

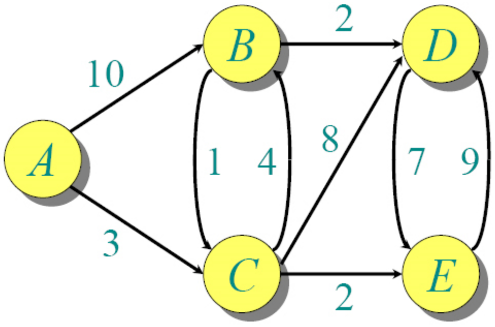
and added to S exactly once, so that the **while** loop of lines 4–8 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy. Chapter 16 explains greedy strategies in detail, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths. The key is to show that each time it adds a vertex u to set S, we have $u.d = \delta(s, u)$.

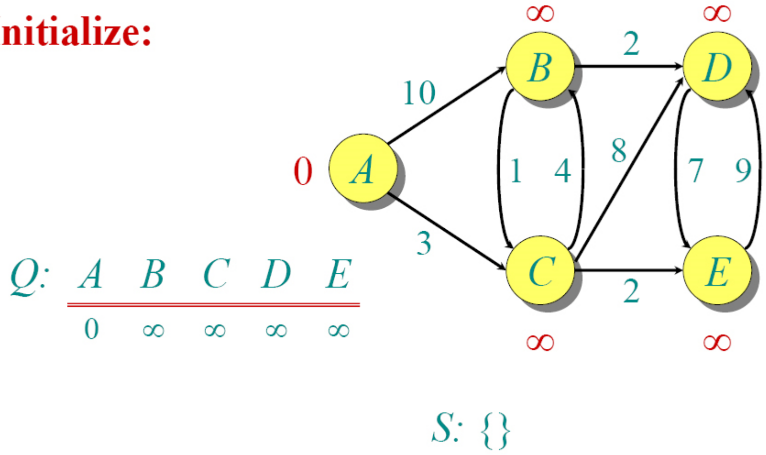
Theorem 24.6 (Correctness of Dijkstra's algorithm)

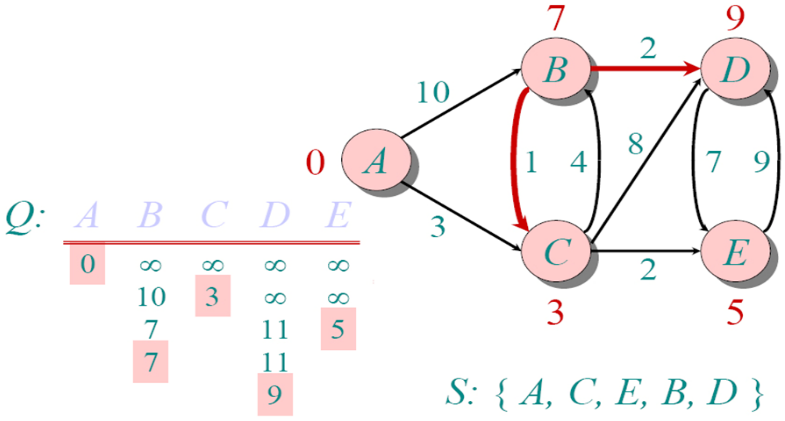
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Example –



Initialize:





V	A	B	C	D	E
A	0	∞	∞	∞	∞
C	0	10	3	∞	∞
E	0	7	3	11	5
B	0	7	3	11	5
D	0	7	3	9	5

SET S
{ A }
{ A , C }
{ A , C , E }
{ A , C , E , B }
{ A , C , E , B , D}

JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d > 0$ and a profit $p > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. In order to complete a job one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. 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. An optimal solution is a feasible solution with maximum value.

Algorithm 4.6 Faster algorithm for job sequencing

```
procedure FJS(D, n, b, J, k)
//find an optimal solution J = J(l), ... , J(k) it is assumed
that
//p1 >= p2 >= ... >= Pm and that b = min{n, max{D(i)}}

integer b, D(n), J(n), F(O:b ), P(O:b)
for i ← 0 to b do //initialize trees
F(i) ← i; P(i) ← - 1
//repeat
k ← 0
for i ← 1 to n do //use greedy rule
j ← FIND(min(n, D(i))
if F(j) != O then k  ← k +1 ;J(k)← i //select job i
L ← FIND(F(j) - 1); call UNION(L,j)
    F(j) ← F(L)  // j may be new root
    Endif
    Repeat
    endFJS
```

Analysis - In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$.

new job being considered there is no α as defined above then it cannot be included in J. The proof of the validity of this statement is left as an exercise.

Example 4.4 Let $n = 5$, $(p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule we have:

J	assigned slots	job being considered	action
ϕ	none	1	assign to [1, 2]
$\{1\}$	[1, 2]	2	assign to [0, 1]
$\{1, 2\}$	[0, 1], [1, 2]	3	cannot fit; reject
$\{1, 2\}$	[0, 1], [1, 2]	4	assign to [2, 3]
$\{1, 2, 4\}$	[0, 1], [1, 2], [2, 3]	5	reject.

The optimal solution is $J = \{1, 2, 4\}$. □

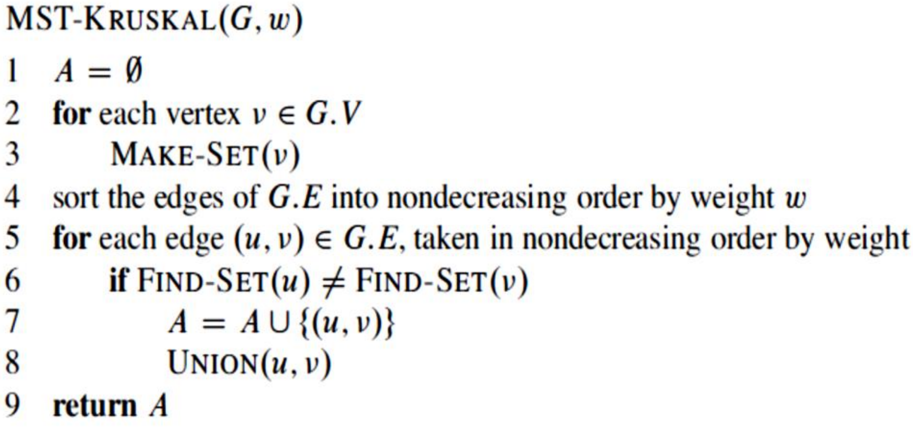
Minimum Spanning Tree –

A tree such as this, which contains every vertex of a connected graph G is a *spanning tree*. Computing a spanning tree Twith smallest total weight is the problem of constructing a *minimum spanning tree* (or *MST*). All the well-known efficient algorithms for finding minimum spanning trees are applications of the *greedy method*. We apply the greedy method by iteratively choosing objects to join a growing collection, by incrementally picking an object that minimizes or maximizes the change in some objective function. The first MST algorithm we discuss is Kruskal's algorithm, which "grows" the MST in clusters by considering edges in order of their weights. The second algorithm we discuss is the Prims algorithm, which grows the MST from a single root vertex, much in the same way as Dijkstra's shortest-path algorithm.

Kruskal's algorithm –

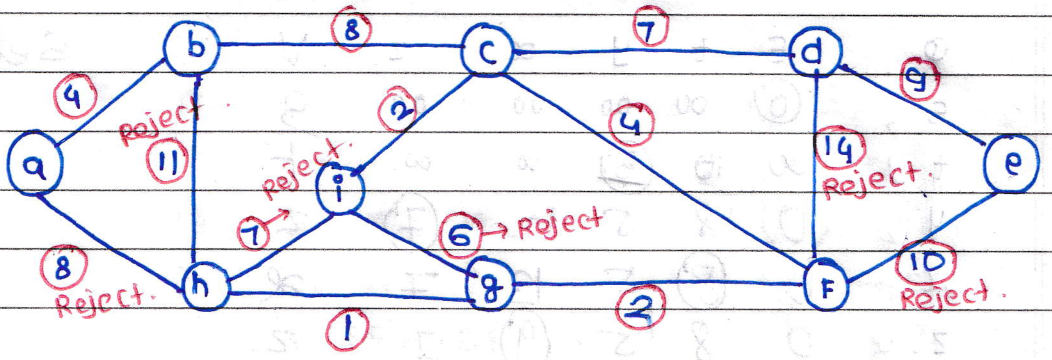
In Kruskal's algorithm, it is used to build the minimum spanning tree in clusters. Initially, each vertex is in its own cluster all by itself. The algorithm then considers each edge in turn, ordered by increasing weight.

- In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.
- Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight. Let C1 and C2 denote the two trees that are connected by (u,v). Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight.

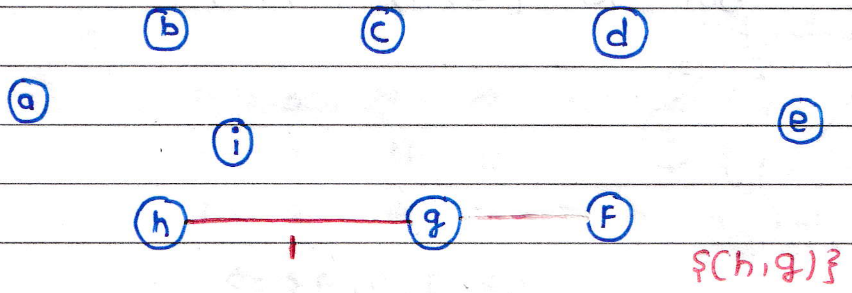


```
MST-KRUSKAL($G, w$)
$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET($v$)
sort the edges of $G.E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
    if FIND-SET($u$) $\neq$ FIND-SET($v$)
        $A = A \cup \{(u, v)\}$
        UNION($u, v$)
return $A$
```

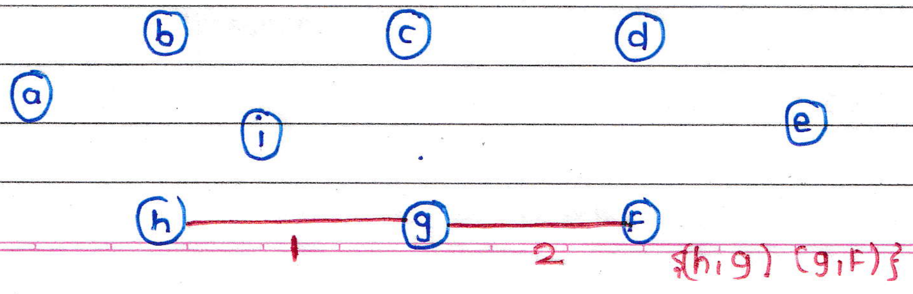
```
MST-KRUSKAL (G,ω)
    A = φ
    For each vertex v ∈ G,V
        MAKE-SET (v)
    Sort the edges of G·E into non-decreasing order
    by weight ω
    For each edge (u,v) ∈ G·E into taken in non-
    decreasing order by weight
        IF FIND-SET (u) ≠ ~~$~~ FIND SET (V)
            A = A ∪ {(u,v)}
            UNION (u,v)
    Return A
```



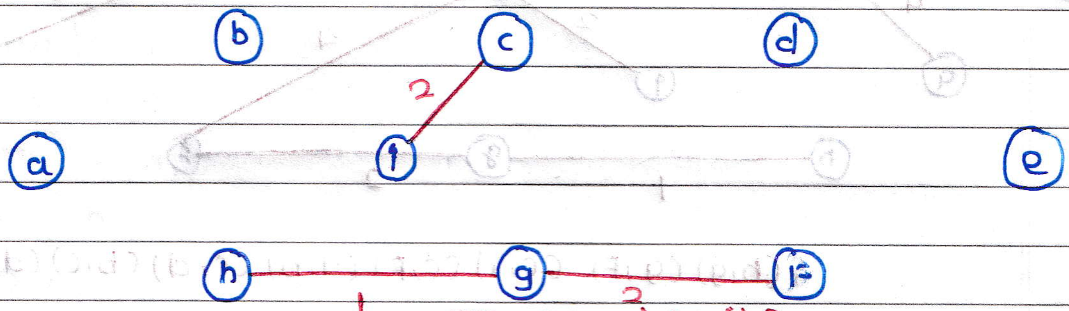
a)



b)

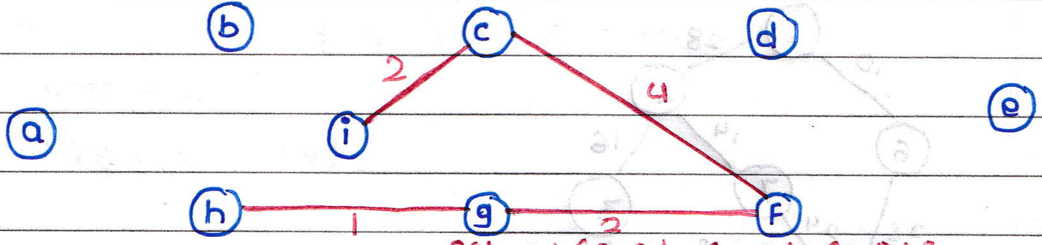


ⓒ



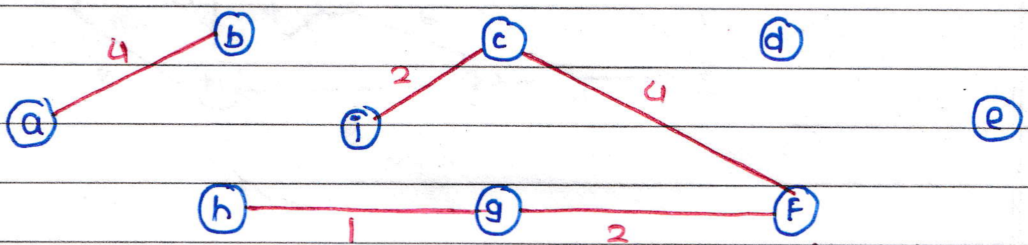
{(h,g) (g,F) (c,i) }

ⓓ



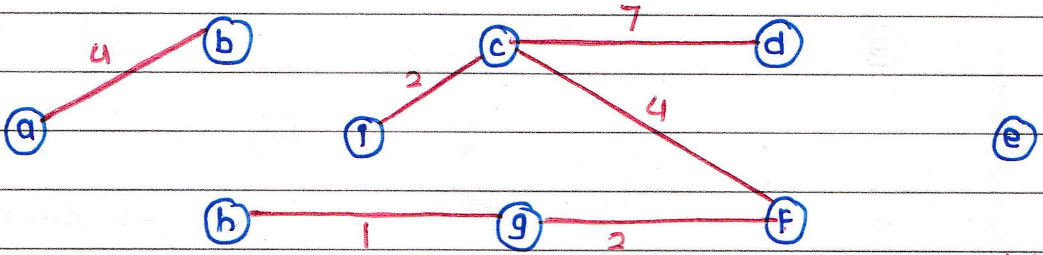
{(h,g) (g,F), (c,i), (c,F)}

ⓔ



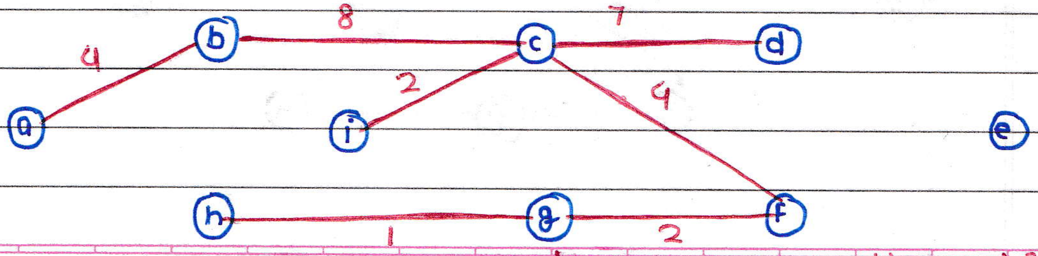
{(h,g) (g,F), (c,i) (cF) (a,b)}

Ⓕ

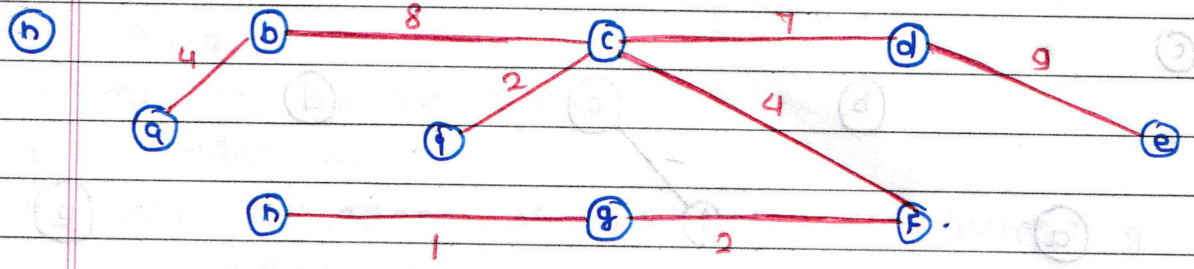


{(h,g)(g,F) (c,i) (C,F) (a,b) (c,d)}

ⓖ



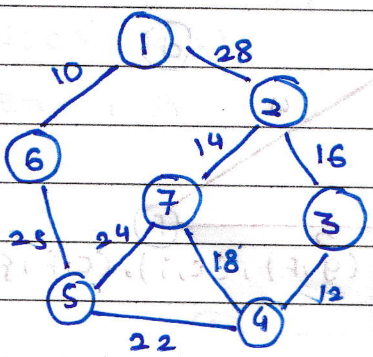
{(h,g) (g,F) (c,i) (c,F) (a,b) (c,d), (b,c)}

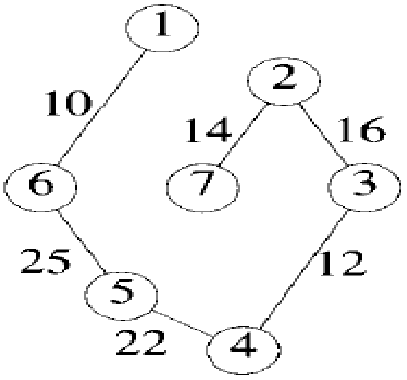


{(h,g) (g,F) (c,i) (c,F) (a,b) (c,d) (b,c) (d,e)}.

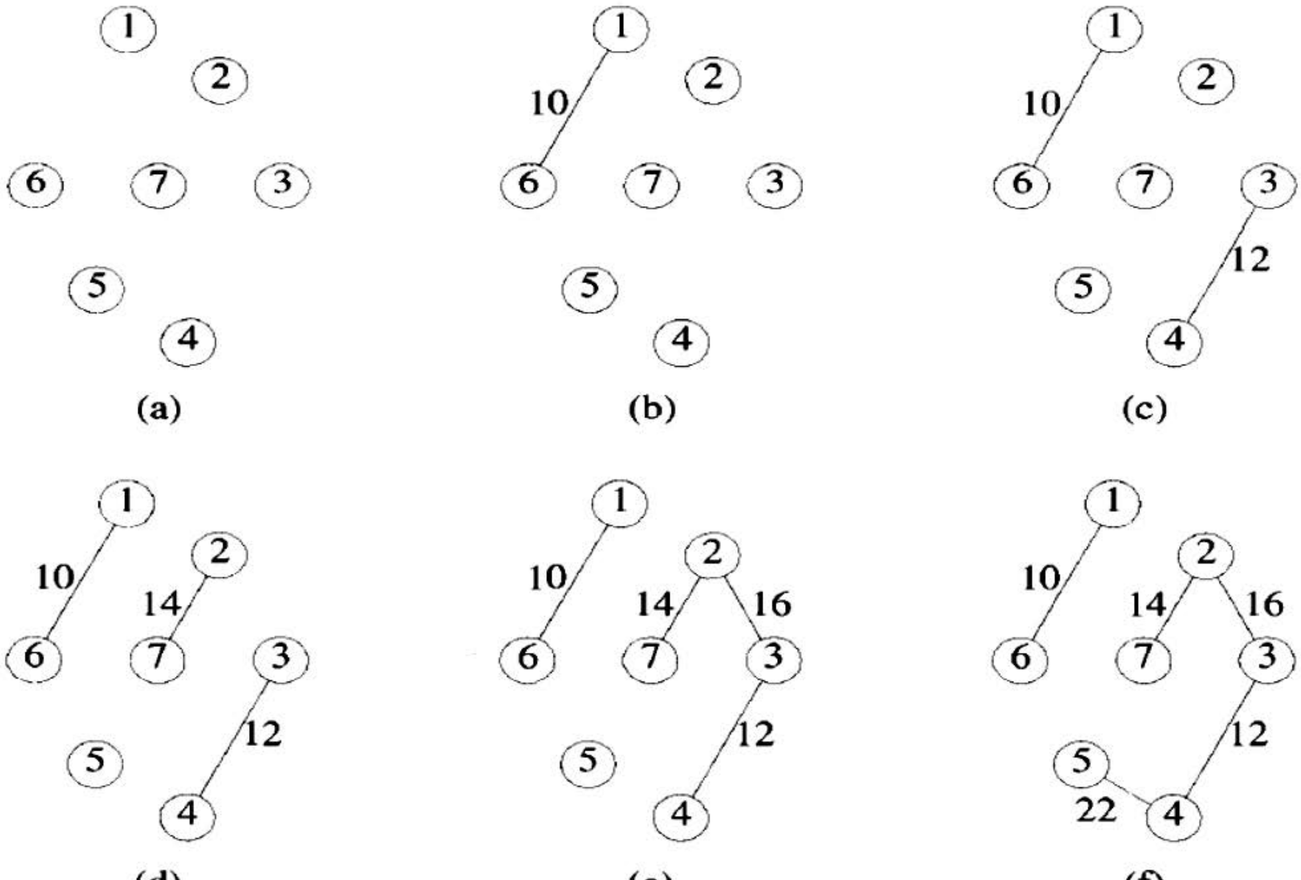
minimum spaning Tree.

ex-2.





(g)

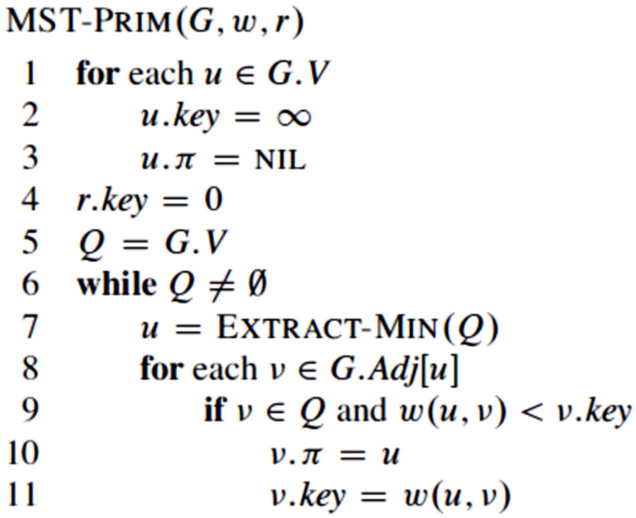


(a) (b) (c)

(d) (e) (f)

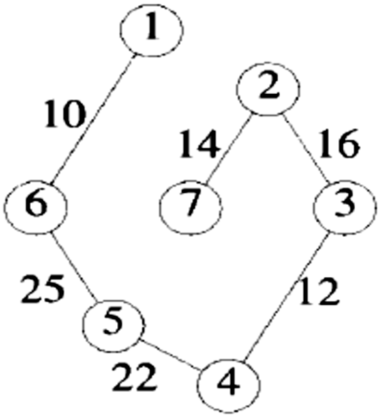
Prims algorithm –

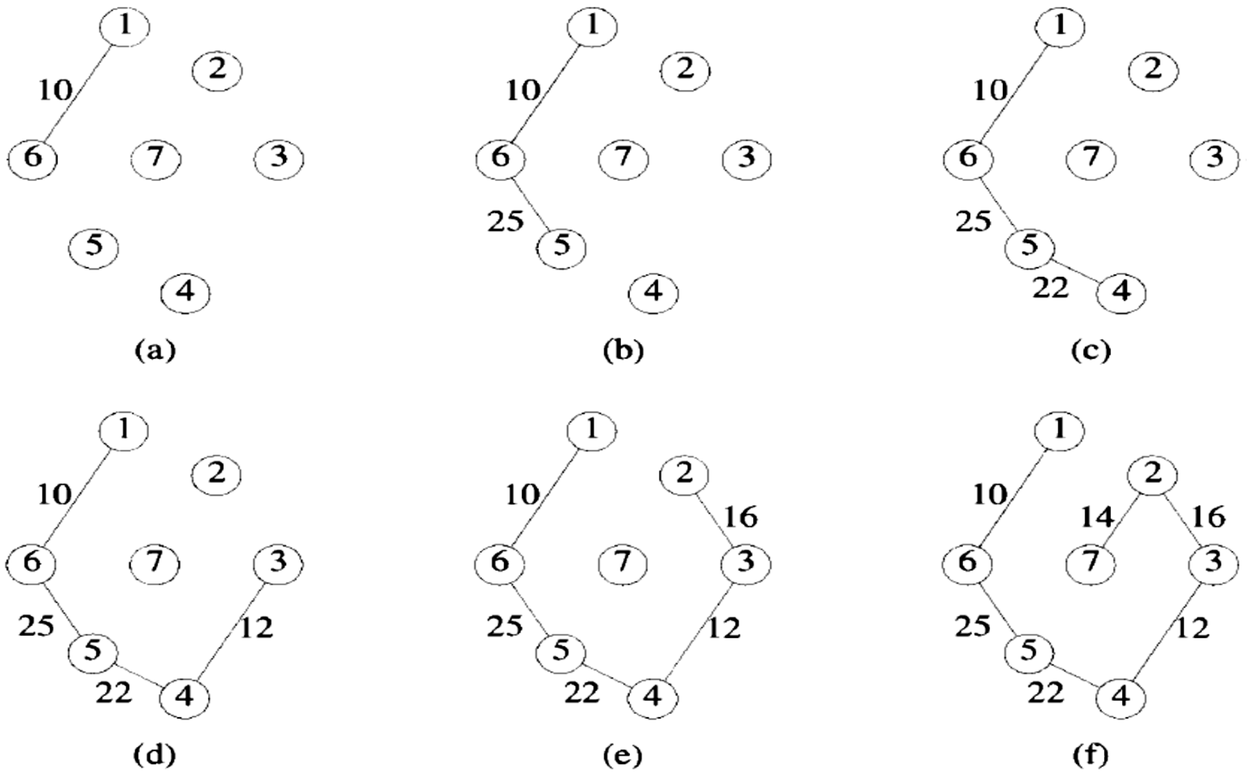
Like Kruskal’s algorithm, Prim’s algorithm is a special case of the generic minimum-spanning-tree method. Prim’s algorithm operates much like Dijkstra’s algorithm for finding shortest paths in a graph. Prim’s algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree A a light edge that connects A to an isolated vertex—one on which no edge of A is incident. This rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree’s weight.



```
MST-PRIM($G, w, r$)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = \text{NIL}$
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

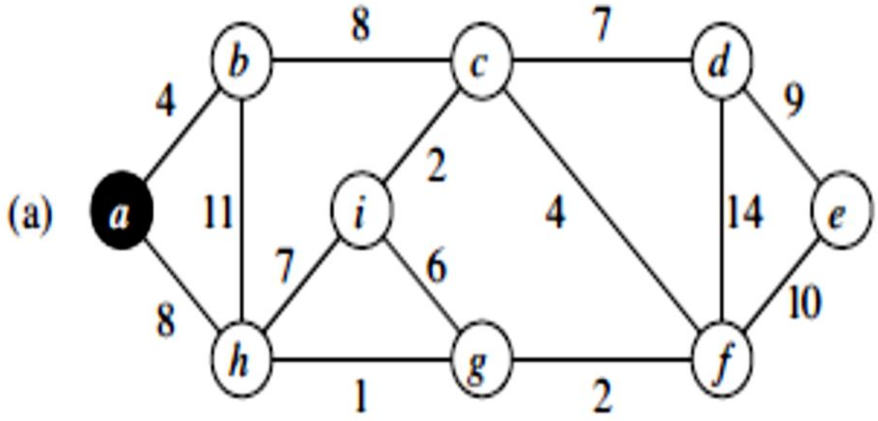
Example 1 –

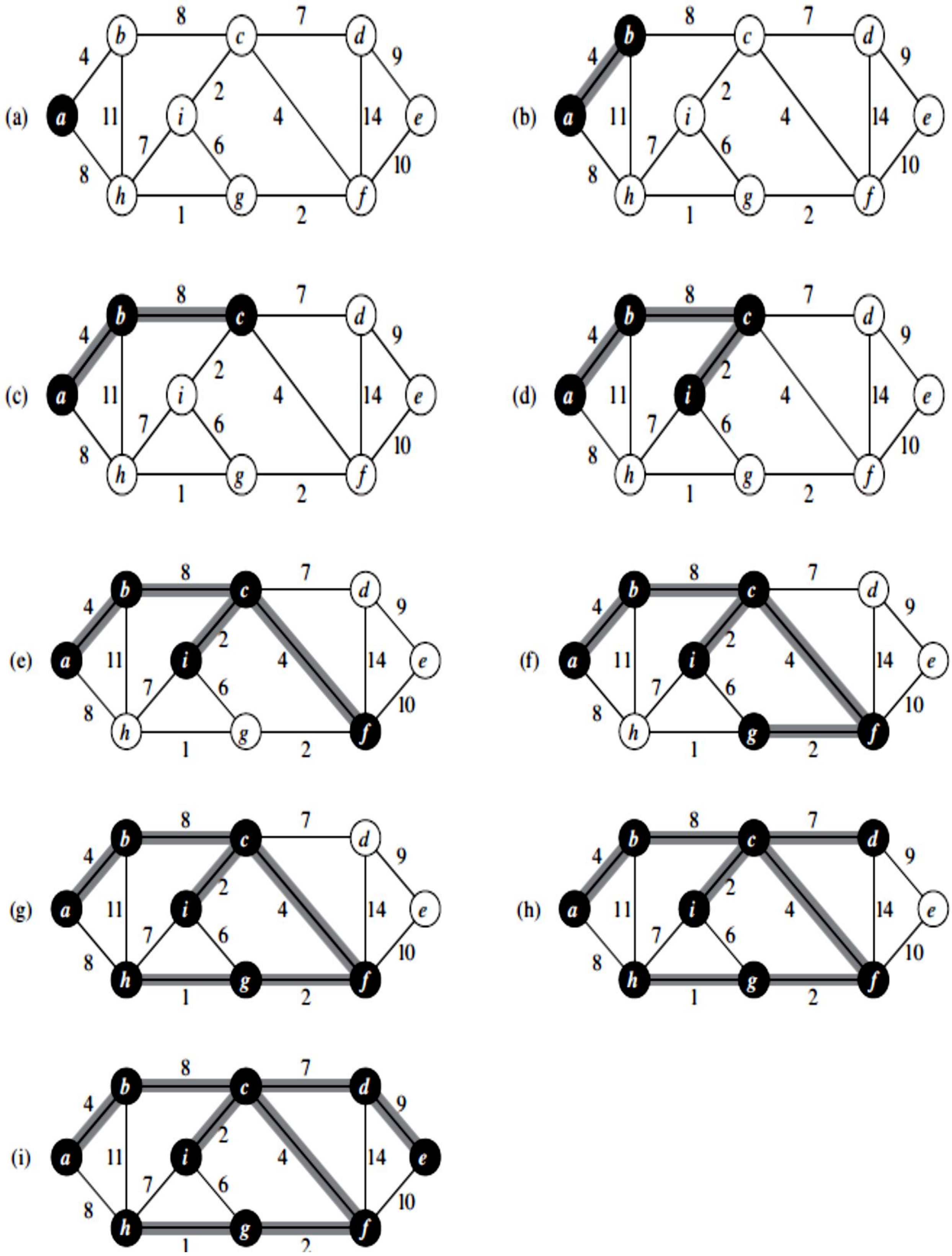




(a) (b) (c) (d) (e) (f)

Example 1 –




(a)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(b)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(c)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(d)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(e)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(f)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(g)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(h)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2
(i)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 10 8 1 2