Unit 6: Greedy Algorithms

Greedy algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - My everyday examples:
    - Driving in Baroda
    - Playing cards
    - Invest on stocks
    - Choose a university
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
- greedy algorithms tend to be easier to code

Why it is Greedy?

- Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled
- The greedy choice is the one that maximizes the amount of unscheduled time remaining

Characteristic of Greedy Algorithm

- ***Solution function*** checks whether a particular set of candidates provides a solution to problem.(ignoring optimal solution)
- ***Feasible function*** checks whether or not it is possible to complete the set by adding further candidates so as to obtain at least one solution
- ***Selection function*** indicates at any time which of the remaining candidates is most promising
- ***Objective function*** gives value of the solution

Characteristic of Greedy Algorithm

- Greedy(C: Candidate set) S← Φ

```
While C != Φ and not solution(S) x← select(C)
        C= C / {x}
        if(feasible(S U{x}) then S← S U {x}
end
If solution(S) return S
else
      return “No Solution found”
end
```

Elements of Greedy Strategy

- An greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen
  - NOT always produce an optimal solution
- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property
  - Optimal substructure

Greedy-Choice Property

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

- Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made.
- 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 sub-problems.

Of course, we must prove that a greedy choice at each step yields a globally optimal solution.

Optimal Substructures

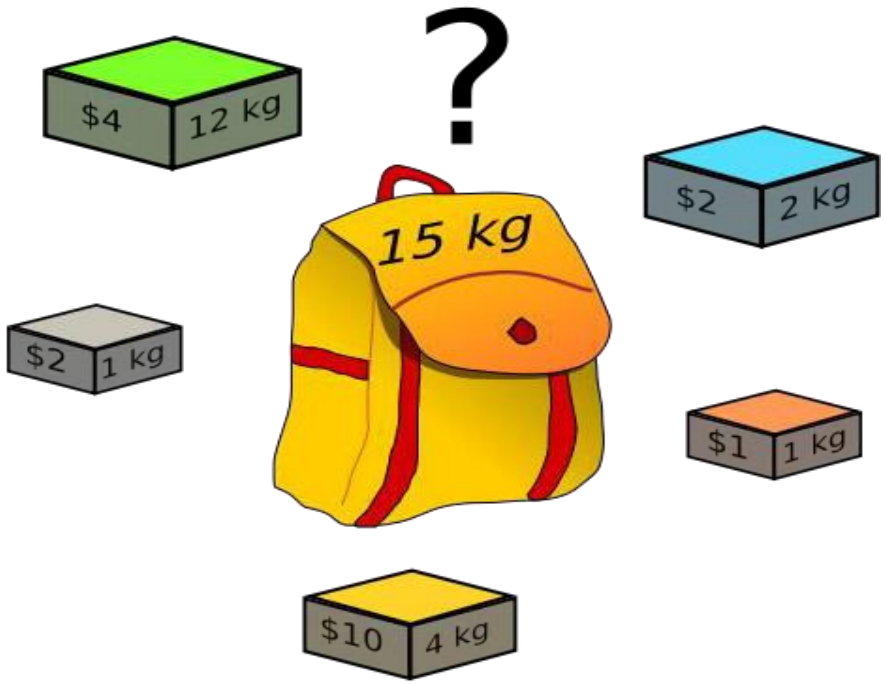
- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems.

Fractional Knapsack Problem

A thief considers taking W Kg of loot.

The loot is in the form of n items, each with weight w_i and value v_i.

Any amount of an item can be put in the knapsack ,weight limit W is not exceeded.



Fractional Knapsack Problem

Item	Value	Weight
1	4	12
2	2	1
3	10	4
4	1	1
5	2	2

Let's select items which gives me highest value

Item 3 Profit =10 Remaining weight is 15-4 = 11

Item 1 Profit =10+4*(11/12) = 10 + 3.66 = 13.66
Remaining weight is 11 -11 =0

Fractional Knapsack Problem

Item	Value	Weight
1	4	12
2	2	1
3	10	4
4	1	1
5	2	2

Let's select items with lowest weight that keeps knapsack empty for long period

Item 4 Profit =1 Remaining weight is 15-1 = 14

Item 2 Profit =1+2 Remaining weight is 14-1 = 13

Item 5 Profit =1+2+2 Remaining weight is 13-2 = 11

Item 3 Profit =1+2+2+10 =15
Remaining weight is 11-4 = 7

Item 1 Profit =15+4*(7/12) = 15 + 2.33 = 17.33
Remaining weight is 7 -7 =0

Fractional Knapsack Problem

Item	Value	Weight	Value/weight
1	4	12	4/12= 0.33
2	2	1	2/1 = 2
3	10	4	10/4 = 2.5
4	1	1	1/1= 1
5	2	2	2/2 = 1

Let's select items with highest profit per unit quantity

Item 3 Profit =10 Remaining weight is 15-4= 11

Item 2 Profit =10+2 Remaining weight is 11-1 = 10

Item 5 Profit =10+2+2 Remaining weight is 10-2 = 8

Item 4 Profit =10+2+2+1 =15
Remaining weight is 8-1 = 7

Item 1 Profit =15+4*(7/12) = 15 + 2.33 = 17.33
Remaining weight is 7 -7 =0

Example: Given $n = 5$ objects and a knapsack capacity $W = 100$ as in Table I. Three solutions???? .

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

Table I

select	x_i					value
Max v_i	0	0	1	0.5	1	146
Min w_i	1	1	1	1	0	156
Max v_i/w_i	1	1	1	0	0.8	164

Table II

There seem to be 3 obvious greedy strategies:

(Max value) Sort the objects from the highest value to the lowest, then pick them in that order.

(Min weight) Sort the objects from the lowest weight to the highest, then pick them in that order.

(Max value/weight ratio) Sort the objects based on the value to weight ratios, from the highest to the lowest, then select.

Greedyknapsack(v[1..n], w[1..n], W)

```
For i= 1 to n
    x[i]=0
    ratio[i]= v[i]/w[i]
end
Weight=0
Profit=0
While (weight <W)
      i= select item from highest ratio to lowest  ratio in order
      if (weight+w[i] <= W )then
                  x[i]=1
                   profit=profit +x[i]*v[i]
                  weight= weight + w[i]
     else
                   x[i]=   (W-weight)/w[i]
                   profit=profit  +x[i]*v[i]
                   weight= W.
     end
end
Return profit.
```

Complexity

Greedy-fractional-knapsack (w, v, W)

- **Input:** an integer *n*, positive values *wi* and *vi* , for $1 \le i \le n$, n_n and another positive value *W*.
- **Output:** *n* values *xi* such that $0 \le xi \le 1$ and $\sum_{i=1}^{n} x_i w_i \le W$ and $\sum_{i=1}^{n} x_i v_i$ is maximized.
- x[i] =0

  weight = 0

  Sort the *n* objects from large to small based on the ratios *vi*/*wi* . The arrays *w*[1..*n*] and *v*[1..*n*] store the respective weights and values after sorting. → $\theta(n \log n)$

  while ($i \le n$ and weight < *W*) → $\theta(n)$

    do i = best remaining item

    IF weight + w[i] $\le$ W

    then x[i] = 1

    weight = weight + w[i]

    i++

    else

    x[i] = (w - weight) / w[i]

    weight = W

    i++

  return x

$$\rightarrow T(n) = \theta(n \log n)$$

- **Time Complexity-**
-

- The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- The average time complexity of **Quick Sort** is O(nlogn).
- Therefore, total time taken including the sort is O(nlogn).

Greedy Algorithm for Fractional Knapsack problem

Fractional knapsack can be solvable by the greedy strategy

- Compute the value per unit weight v_i/w_i for each item
- Obeying a greedy strategy, take as much as possible of the item with the greatest value per unit weight.
- If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per unit weight, and so forth until there is no more room
- $O(n \lg n)$ (we need to sort the items by value per unit)

PRACTICE PROBLEM (FRACTIONAL KNAPSACK PROBLEM)

- Find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-
- n = 5
- w = 60 kg
- (w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)
- (b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)

Job Scheduling Problem

Job Sequencing With Deadlines

The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.

Here-

- You are given a set of jobs.
- Each job has a defined deadline and some profit associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs.
- Processor takes one unit of time to complete a job.

The problem states-

“How can the total profit be maximized if only one job can be completed at a time?”

Approach to Solution

- ✓ A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
- ✓ Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
- ✓ An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Greedy Algorithm

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.
The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

Step-01:

Sort all the given jobs in decreasing order of their profit.

Step-02:

- Check the value of maximum deadline.
- Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-03:

- Pick up the jobs one by one.
- Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Greedy Algorithm

Problem-

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Answer the following questions-

1.Write the optimal schedule that gives maximum profit.

2.Are all the jobs completed in the optimal schedule?

3.What is the maximum earned profit?

Greedy Algorithm

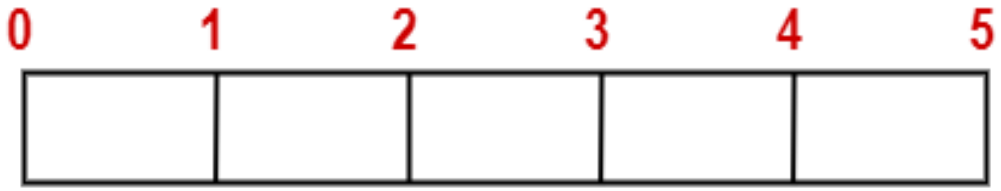
Solution-

Step-01: Sort all the given jobs in decreasing order of their profit-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.
So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



Gantt Chart

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

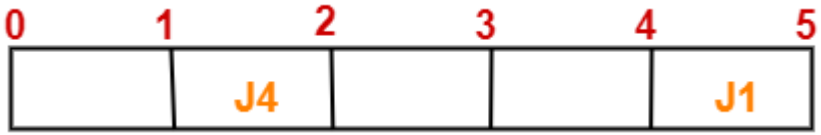
Greedy Algorithm

Step-03:

•We take job J4.
•Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-

Step-04:

•We take job J1.
•Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Step-05:

•We take job J3.
•Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-

Greedy Algorithm

Step-06:

•We take job J2.
•Since its deadline is 3, so we place it in the first empty cell before deadline 3.
•Since the second and third cells are already filled, so we place job J2 in the first cell as-

Step-07:

•Now, we take job J5.
•Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-

Now,
•The only job left is job J6 whose deadline is 2.
•All the slots before deadline 2 are already occupied.
•Thus, job J6 can not be completed.

Solution – Job Scheduling Problem

Part-01:

The optimal schedule is:- **J2 , J4 , J3 , J5 , J1**
This is the required order in which the jobs must be completed in order to obtain the maximum profit.

Part-02:

•All the jobs are not completed in optimal schedule.
•This is because job J6 could not be completed within its deadline.

Part-03:

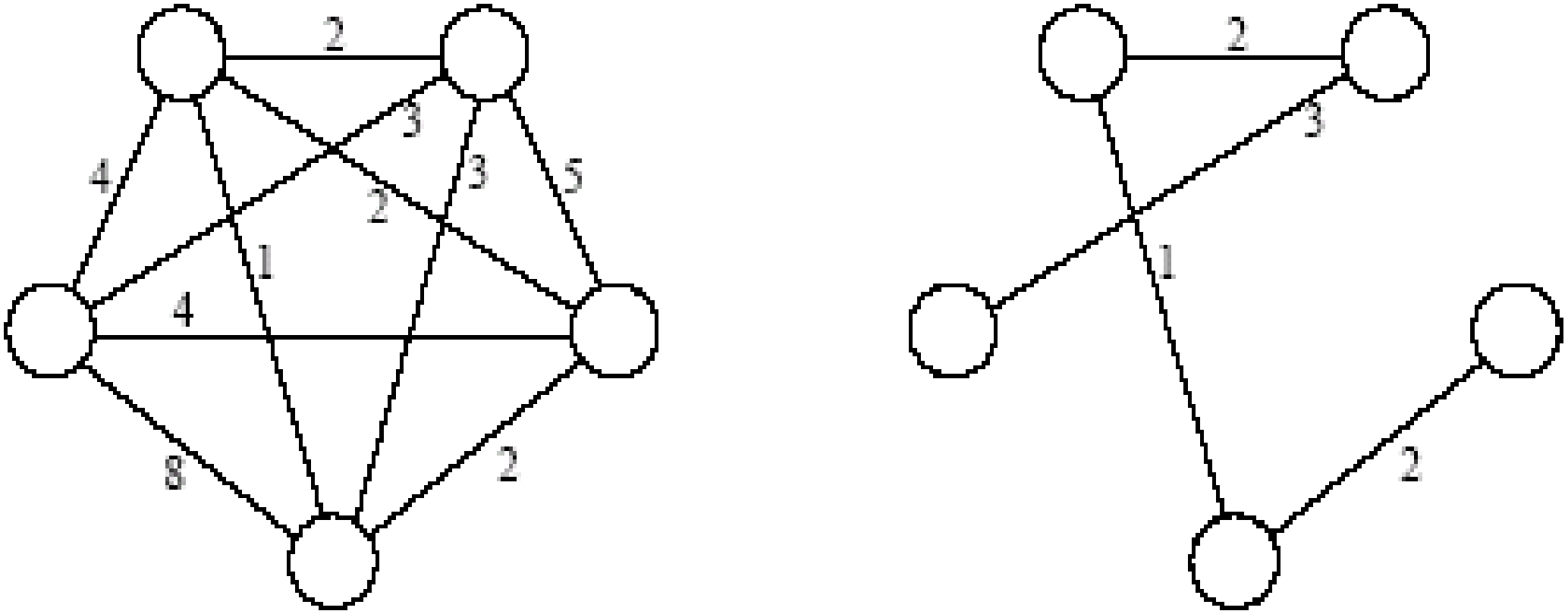
Maximum earned profit
= Sum of profit of all the jobs in optimal schedule
= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1
= 180 + 300 + 190 + 120 + 200
= 990 units

Kruskal’s and Prim’s algorithms

Minimum Spanning Tree

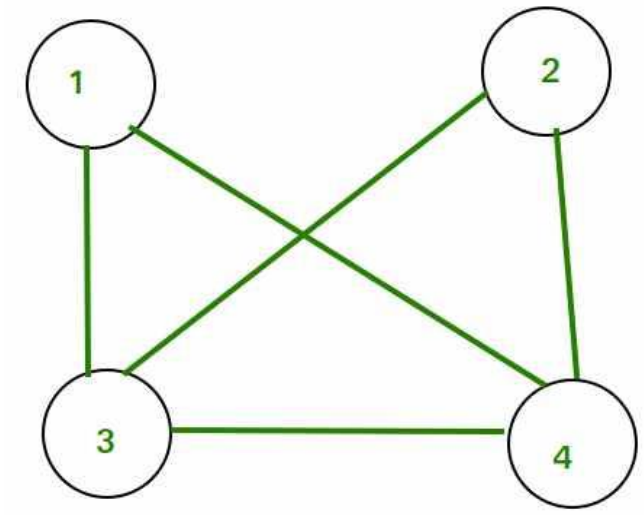
- A *spanning tree* of an undirected graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*.
- In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.
- A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight.

Minimum Spanning Tree



An undirected graph and its minimum spanning tree.

Kirchoff Method



Adjacency Matrix for the above graph will be as follows:

	1	2	3	4
1	0	0	1	1
2	0	0	1	1
3	1	1	0	1
4	1	1	1	0

After applying STEP 2 and STEP 3 – write degree of node diagonal adjacency matrix will look like

	1	2	3	4
1	2	0	-1	-1
2	0	2	-1	-1
3	-1	-1	3	-1
4	-1	-1	-1	3

Cofactor of any element aij=(−1)^(i+j)|Mji|

$$\tilde{a}_{11} = (-1)^{1+1}|M_{11}|$$
$$= a_{22}a_{33}a_{44} + a_{23}a_{34}a_{42} + a_{24}a_{32}a_{43}$$
$$- a_{24}a_{33}a_{42} - a_{23}a_{32}a_{44} - a_{22}a_{34}a_{43}$$

A complete undirected graph can have maximum n^(n-2) number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence 3^(3−2) = 3 spanning trees are possible.

General Properties of Spanning Tree:

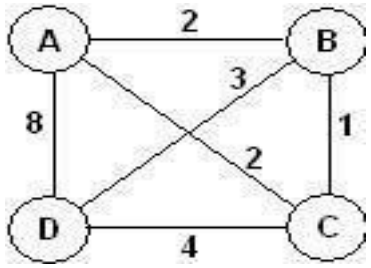
1. A connected graph G can have more than one spanning tree.
2. All possible spanning trees of graph G, have the same number of edges and vertices.
3. The spanning tree does not have any cycle (loops).
4. Spanning tree has n-1 edges, where n is the number of nodes (vertices).
5. A complete graph can have maximum n(n-2) number of spanning trees.

Application of Spanning Tree:

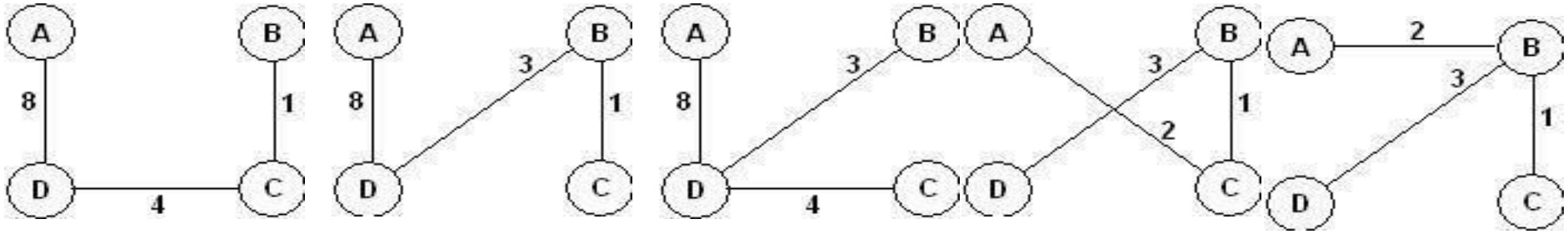
1. Civil Network Planning
2. Computer Network Routing Protocol
3. Cluster Analysis

Minimum Spanning Tree

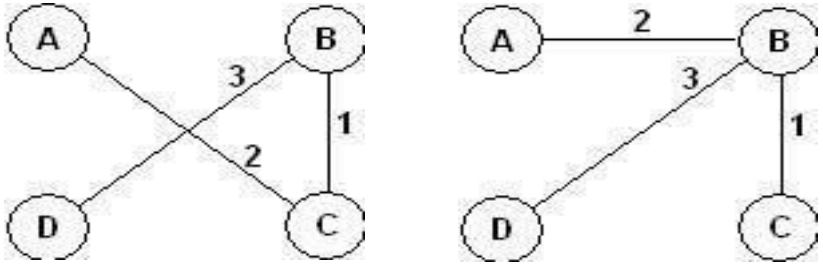
- Example: The graph



Has 16 spanning trees. Some are:



The graph has two minimum-cost spanning trees, each with a cost of 6:



We **are interested in:**

Finding a tree T that contains all the vertices of graph G ➔Spanning tree

And has the least total weight over all such tree ➔

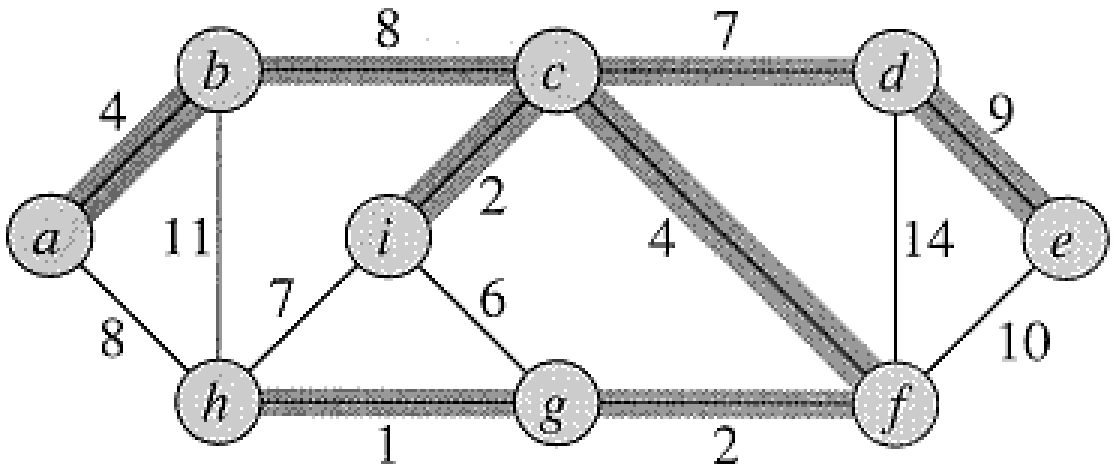
minimum-spanning tree

(MST)

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

Minimum Spanning Trees

- Example Problem
  - You are planning a new telecommunications network to connect a number of remote mountain villages in a developing country.
  - The cost of building a link between pairs of neighbouring villages *(u,v)* has been estimated: *w(u,v).*
  - You seek the minimum cost design that ensures each village is connected to the network.
  - The solution is called a *minimum spanning tree (MST)*.



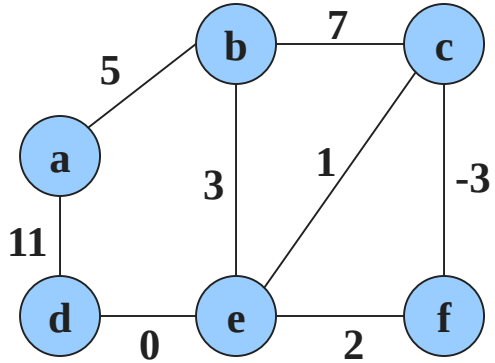
Greedy Spanning Tree Algorithm

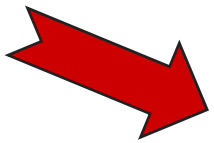
- Spanning tree algorithm is as follows:
- Examine the edges in graph in any arbitrary sequence.
- Decide whether each edge will be included in the spanning tree.
- Note that each time a step of the algorithm is performed, one edge is examined. If there is only a finite number of edges in the graph, the algorithm must halt after a finite number of steps.
- Thus, the time complexity of this algorithm is clearly O(n), where n is the number of edges in the graph.

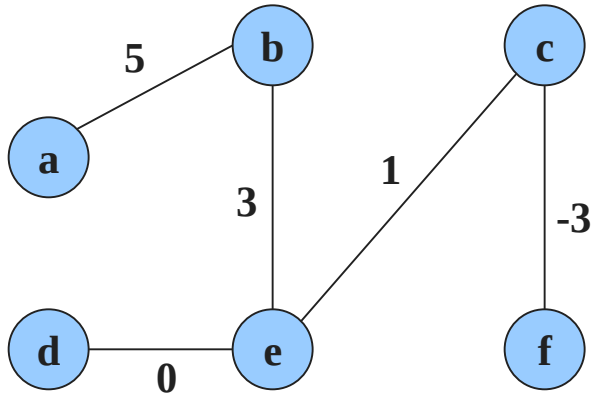
Minimum Spanning Trees

- **Given:** Connected, undirected, weighted graph, *G*
- **Find:** Minimum - weight spanning tree, *T*
- **Example:**

Acyclic subset of edges(*E*) that connects all vertices of *G*.







Kruskal's Algorithm

- Kruskal's Algorithm for computing the minimum spanning tree is directly based on the generic MST algorithm.

- Algorithm consider each edge in turn, order by increasing weight.

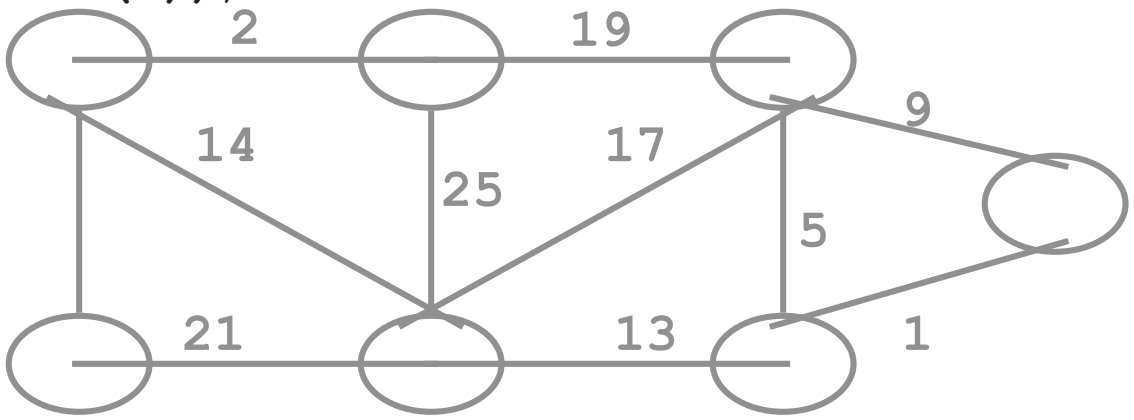
- If an edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST, and two trees connected by an edge (u, v) are merged into a single tree

- If an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

1 Sort

O(V) MakeSet() calls

O(E) FindSet() calls

O(V) Union() calls

Analysis

- Initialize *A*: *O(*1*)*
- First **for** loop: |*V*| MAKE-SETs
- Sort *E*: *O(E* lg *E)*
- Second **for** loop: *O(E)* FIND-SETs and UNIONs
- Assuming the implementation of disjoint-set data structure.

$O((V + E)\,\alpha(V)) + O(E \lg E)$.

Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.

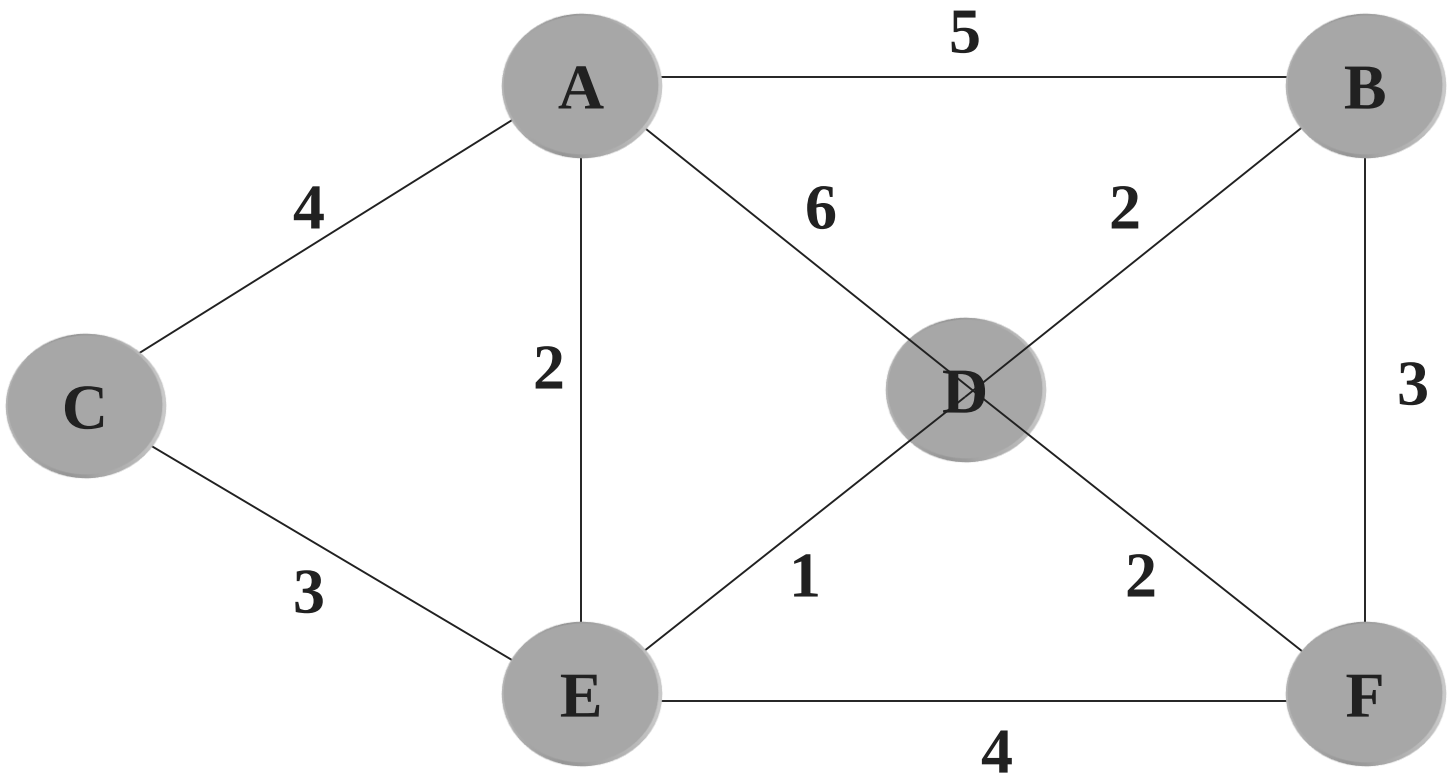
$\alpha(|V|) = O(\lg V) = O(\lg E)$.

Therefore, total time is $O(E \lg E)$.

$|E| \leq |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.

Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\,\alpha(V))$, which is almost linear.)

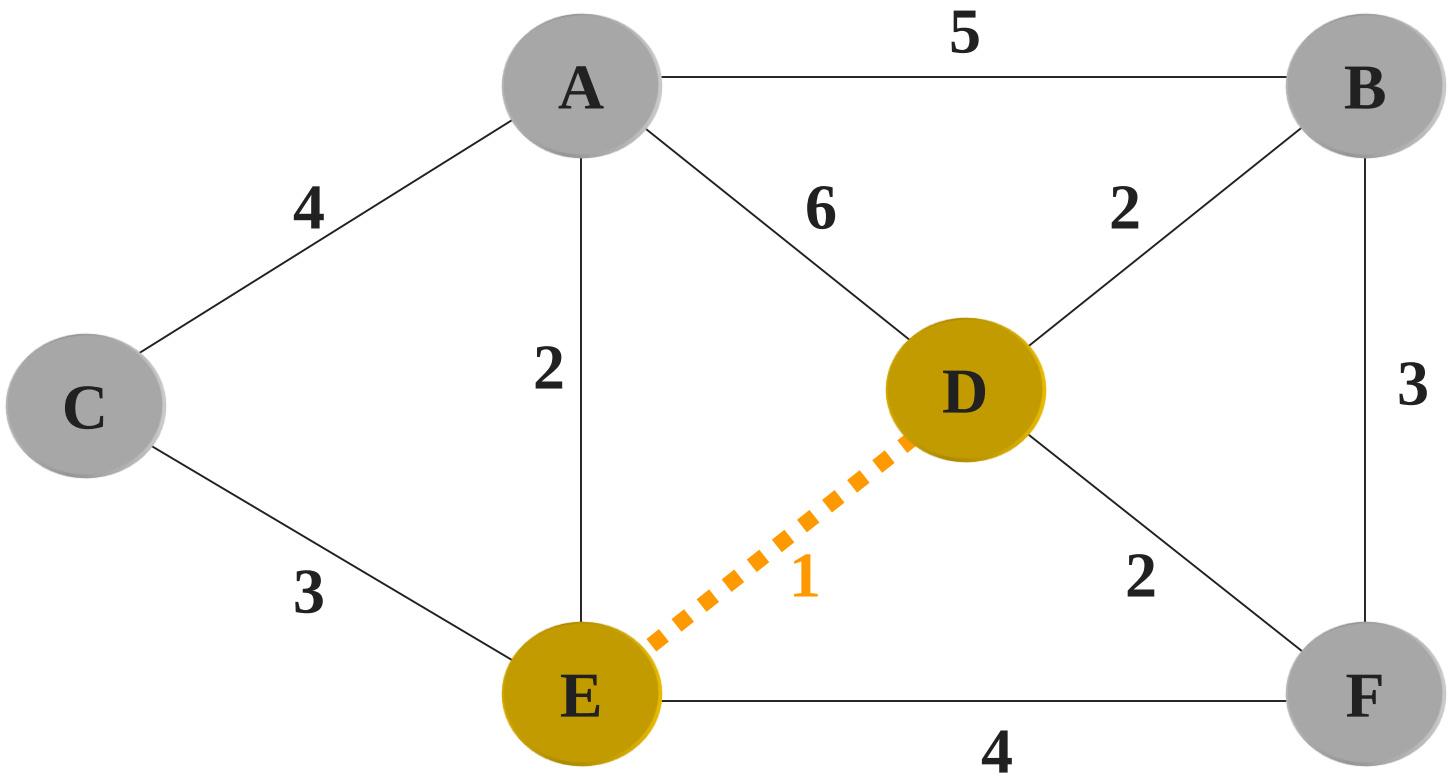
Example



INITIALIZATION

- Step Edge Connected
  Considered Components
  Init - {A}{B}{C}{D}{E}{F}

Example



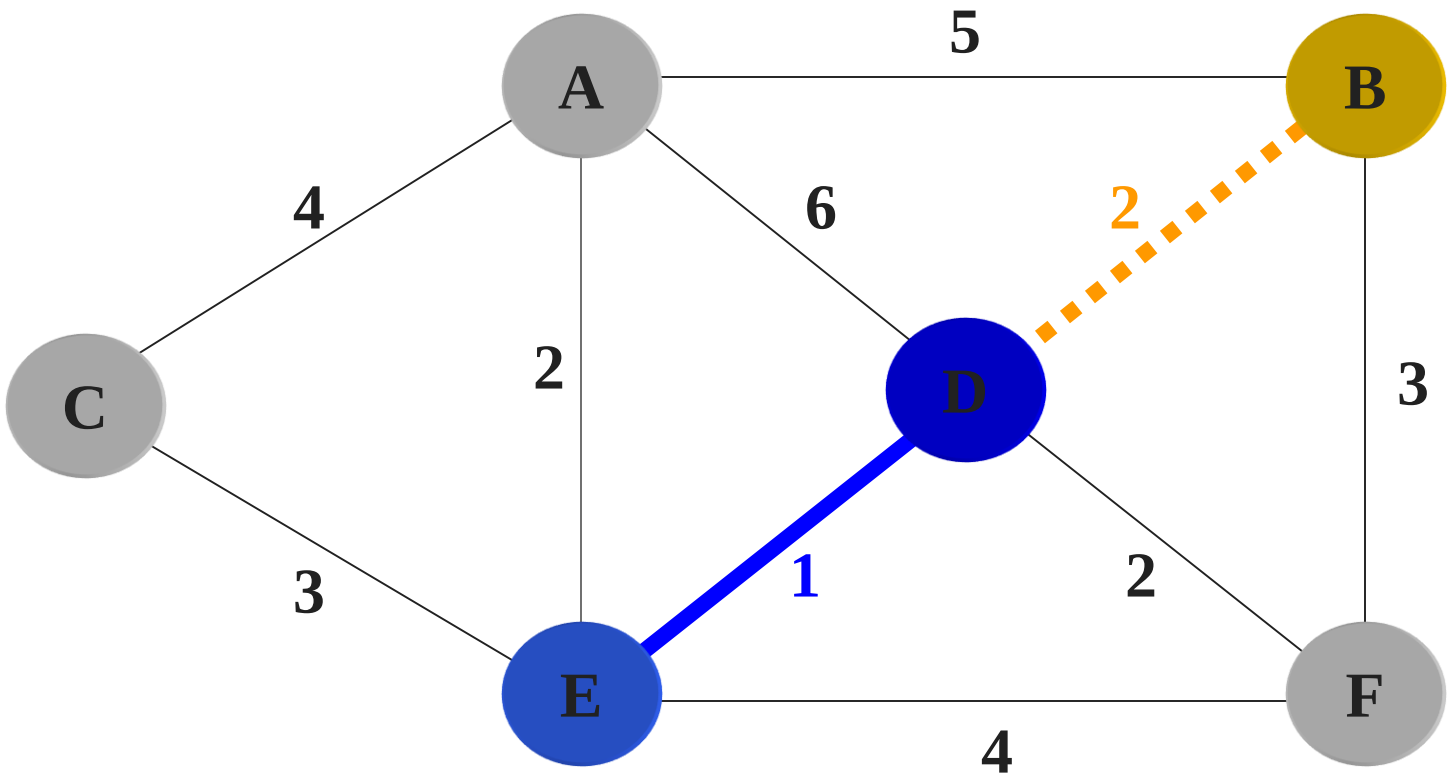
Example

- Step Edge Considered Connected Components

  Init - {A}{B}{C}{D}{E}{F}

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

Example

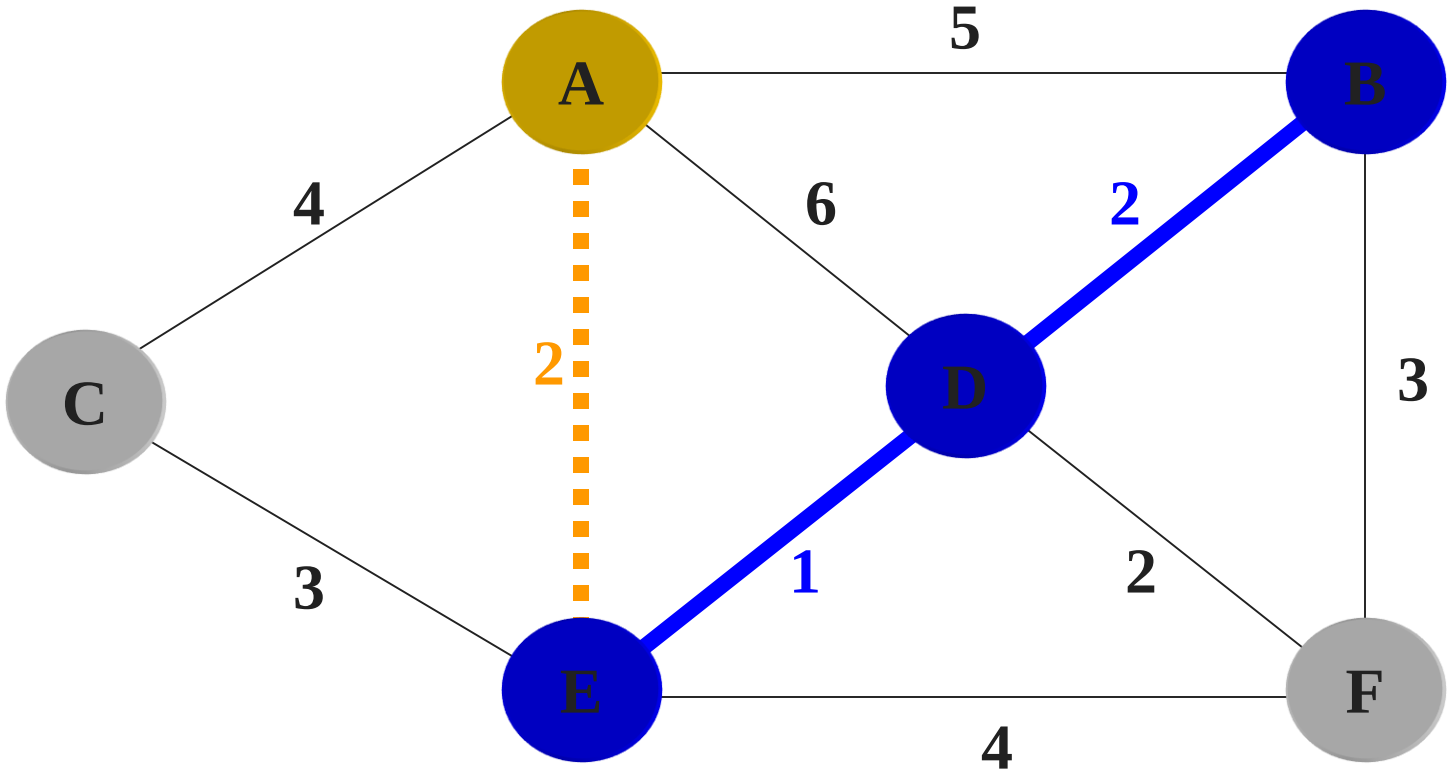


Example

- Step Edge Connected

	Considered	Components
Init	-	{A}{B}{C}{D}{E}{F}
1	{E,D}	{A},{B},{C},{D,E},{F}
2	{D,B}	{A},{B,D,E},{C},{F}

Example

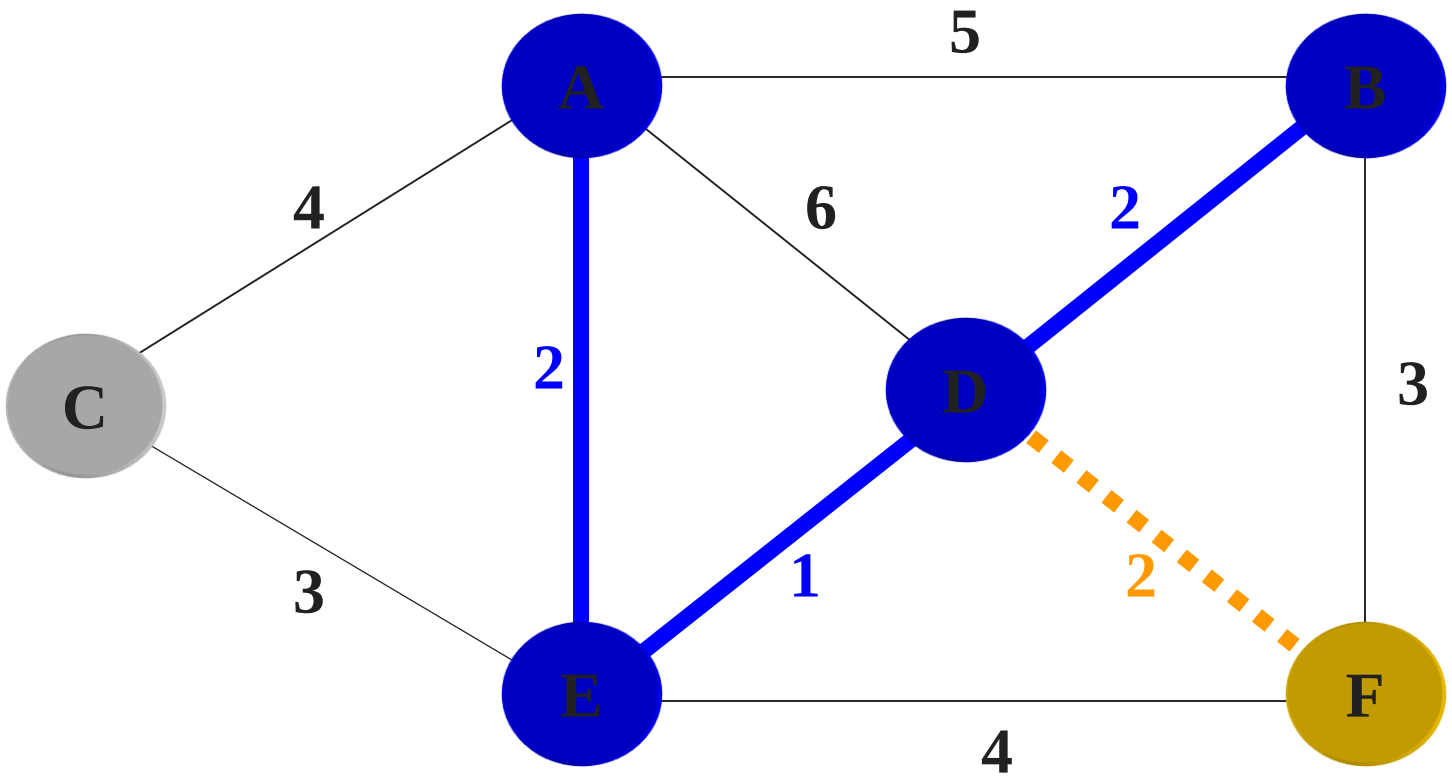


Example

- Step Edge Considered Connected Components

Step	Edge Considered	Connected Components
Init	-	{A}{B}{C}{D}{E}{F}
1	{E,D}	{A},{B},{C},{D,E},{F}
2	{D,B}	{A},{B,D,E},{C},{F}
3	{E,A}	{A,B,D,E},{C},{F}

Example

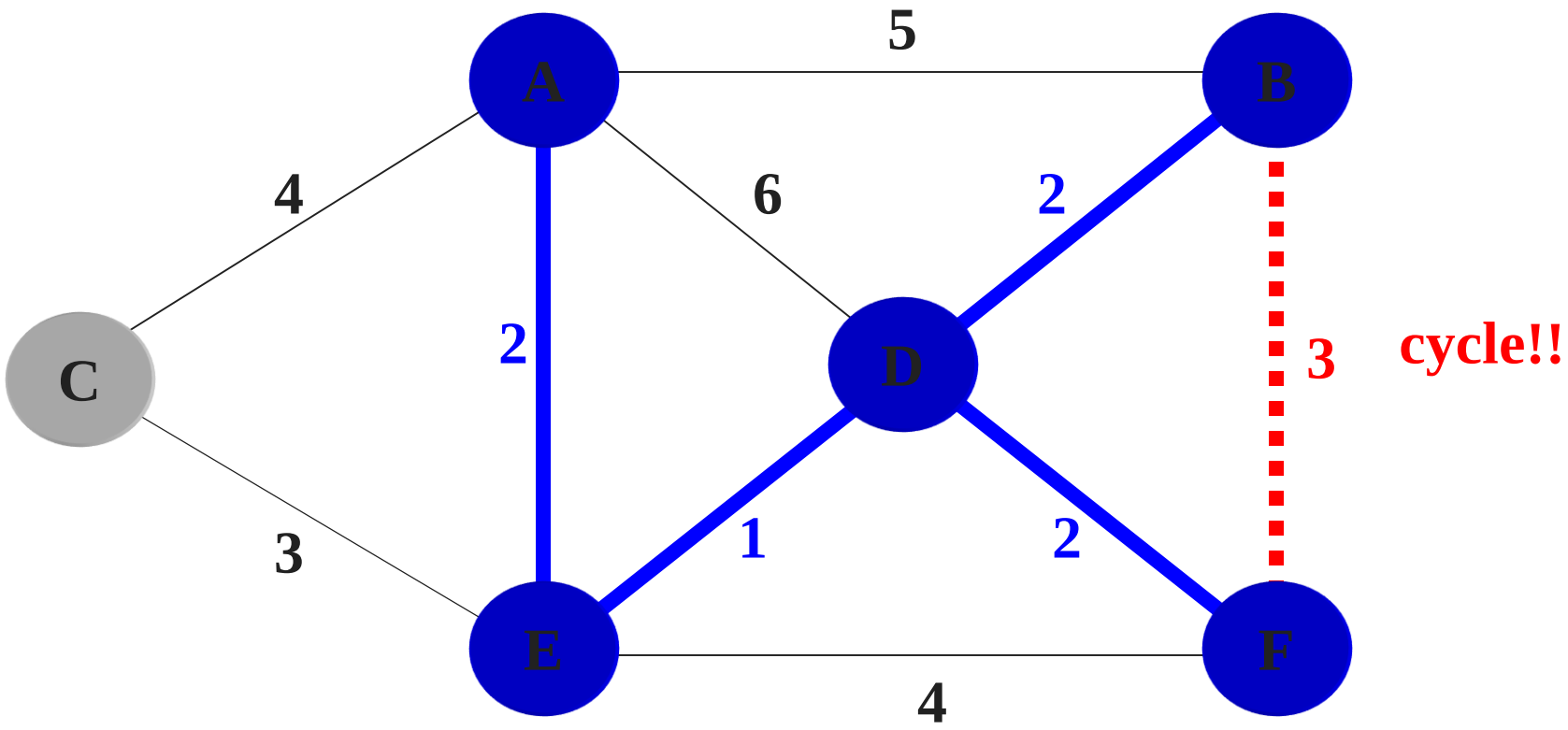


Example

- Step Edge Considered Connected Components

Step	Edge Considered	Connected Components
Init	-	{A}{B}{C}{D}{E}{F}
1	{E,D}	{A},{B},{C},{D,E},{F}
2	{D,B}	{A},{B,D,E},{C},{F}
3	{E,A}	{A,B,D,E},{C},{F}
4	{D,F}	{A,B,D,E,F},{C}

Example

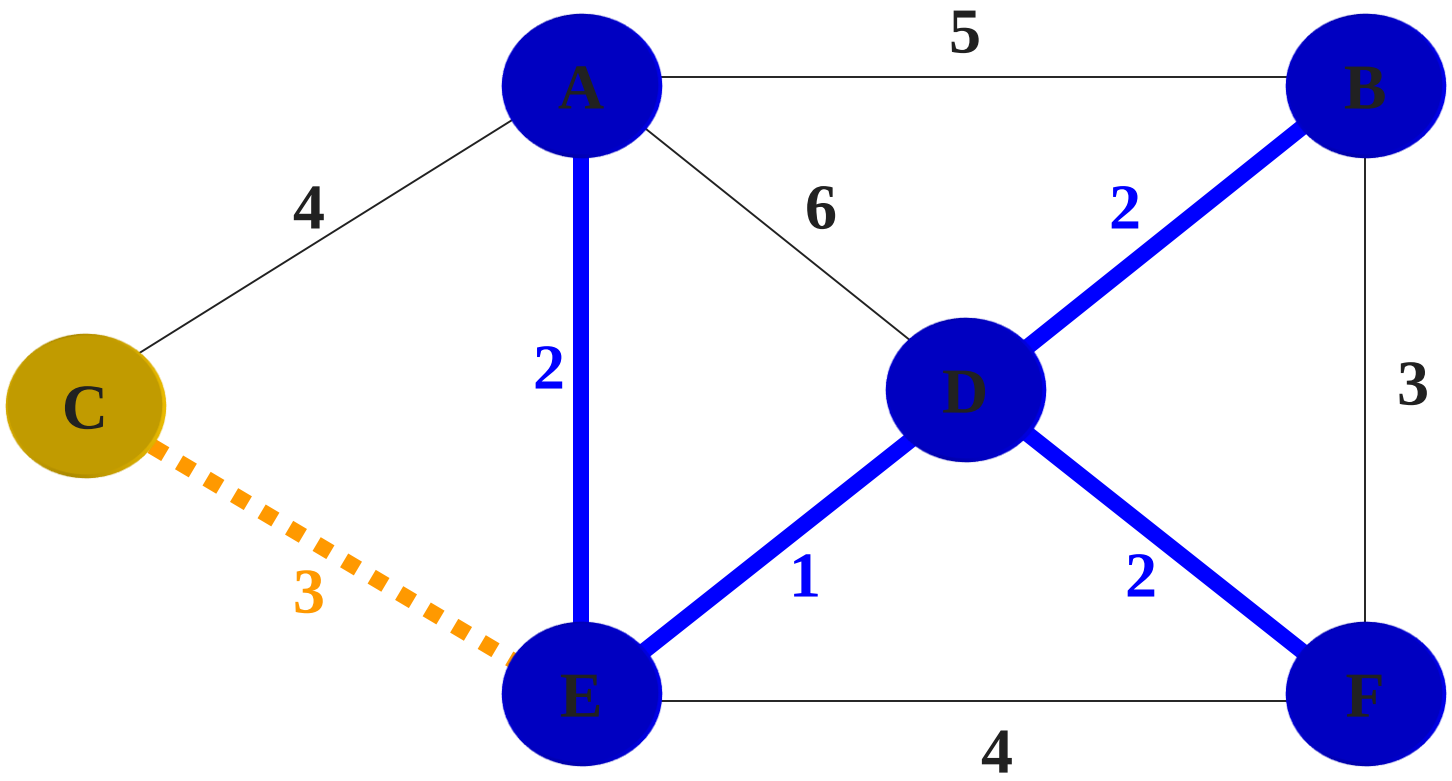


Example

- Step Edge Connected

	Considered	Components
Init	-	{A}{B}{C}{D}{E}{F}
1	{E,D}	{A},{B},{C},{D,E},{F}
2	{D,B}	{A},{B,D,E},{C},{F}
3	{E,A}	{A,B,D,E},{C},{F}
4	{D,F}	{A,B,D,E,F},{C}
5	{F,B}	REJECTED

Example



Example

- Step Edge Considered Connected Components

Step	Edge Considered	Connected Components
Init	-	{A}{B}{C}{D}{E}{F}
1	{E,D}	{A},{B},{C},{D,E},{F}
2	{D,B}	{A},{B,D,E},{C},{F}
3	{E,A}	{A,B,D,E},{C},{F}
4	{D,F}	{A,B,D,E,F},{C}
5	{F,B}	REJECTED
6	{E,C}	{A,B,C,D,E,F}

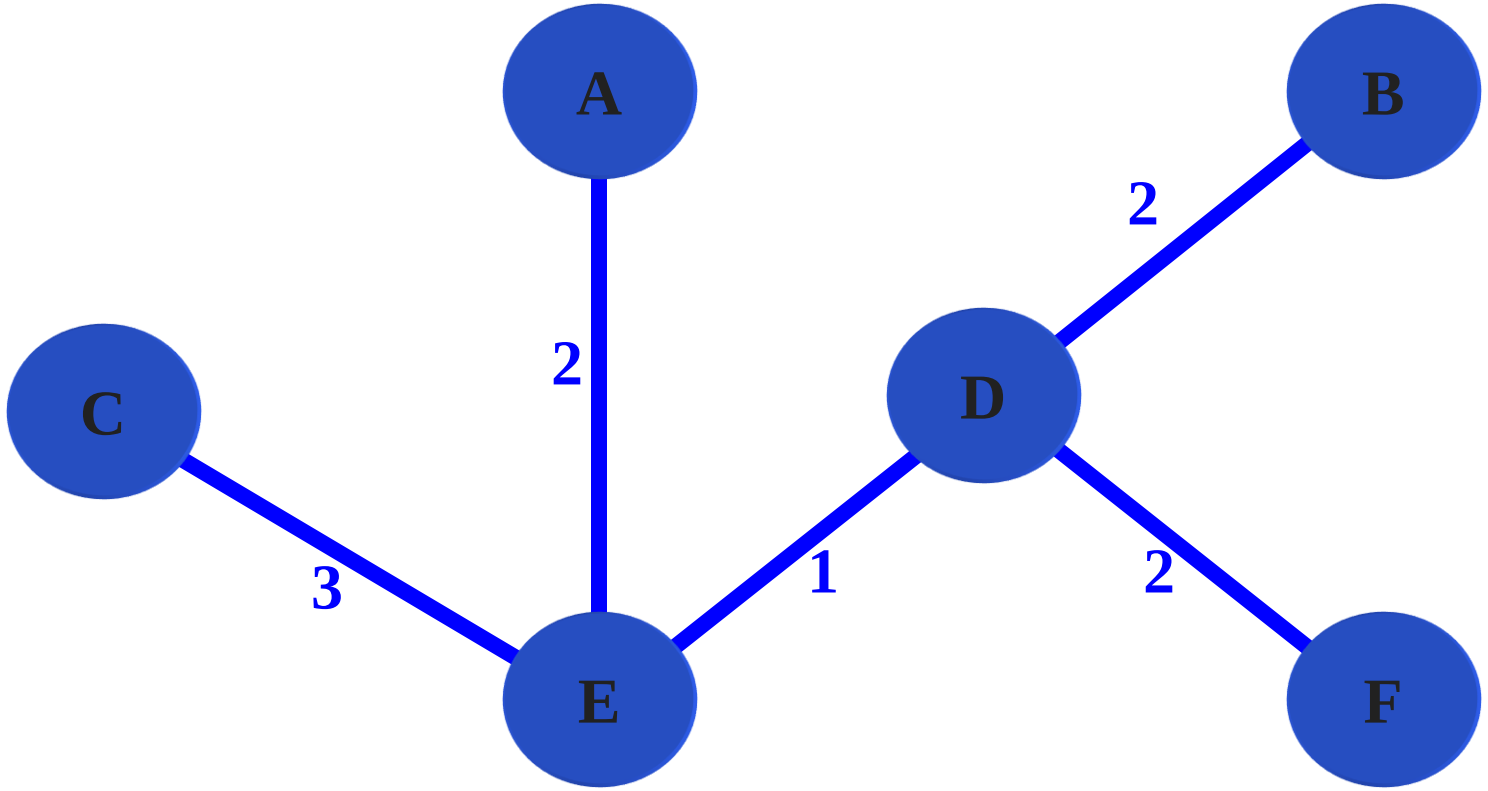
Example

- Step Edge Considered Connected Components

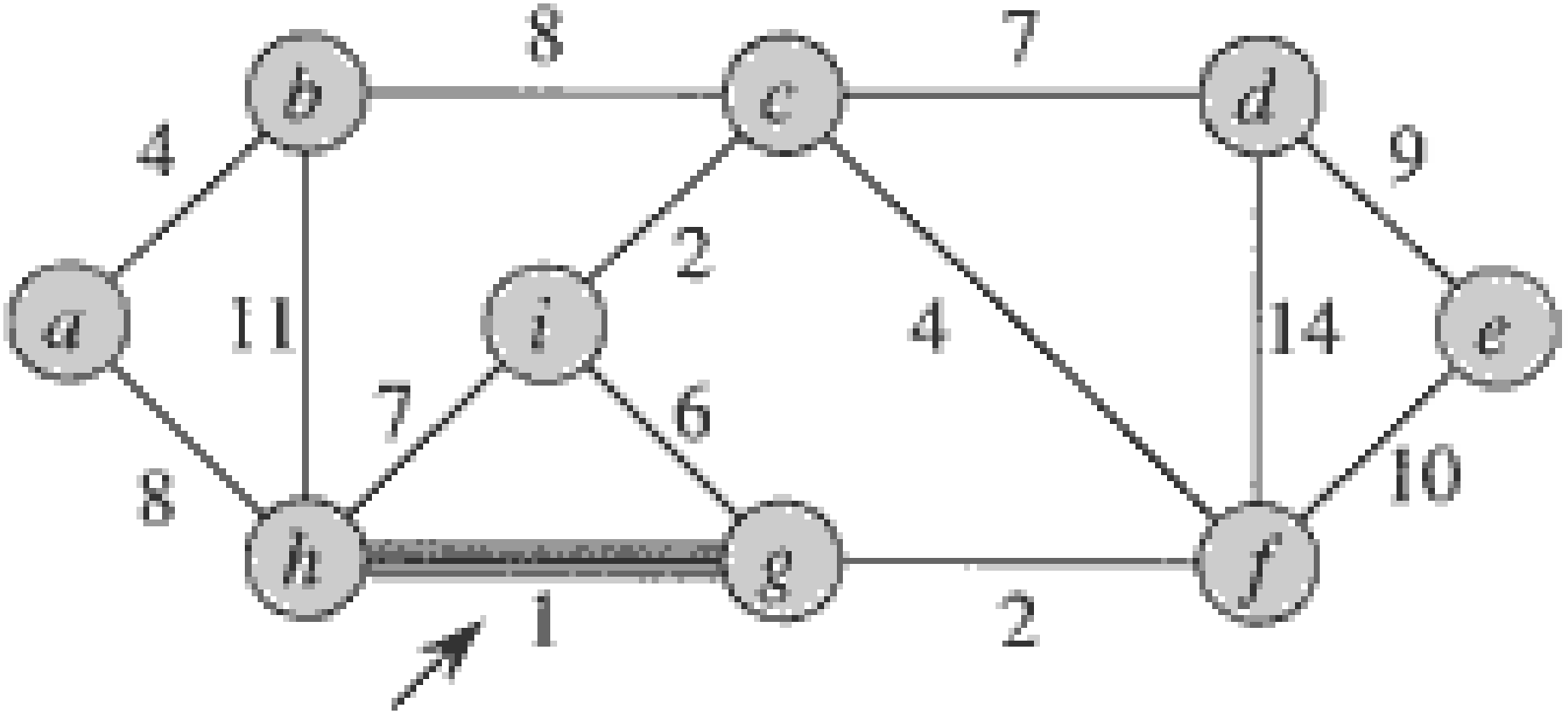
Step	Edge Considered	Connected Components
Init	-	{A}{B}{C}{D}{E}{F}
1	{E,D}	{A},{B},{C},{D,E},{F}
2	{D,B}	{A},{B,D,E},{C},{F}
3	{E,A}	{A,B,D,E},{C},{F}
4	{D,F}	{A,B,D,E,F},{C}
5	{F,B}	REJECTED
6	{E,C}	{A,B,C,D,E,F}

Example

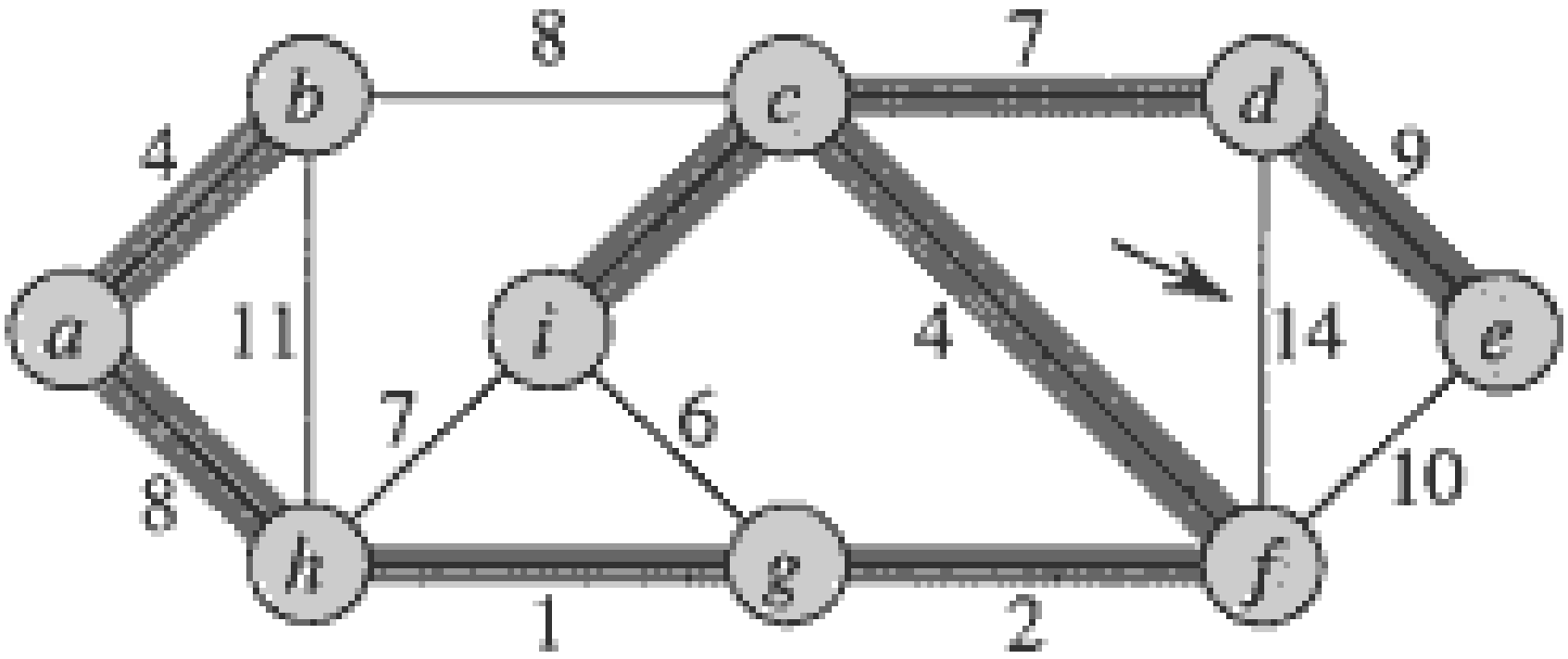
minimum- spanning tree



Kruskal's Algorithm: Example



Kruskal's Algorithm: Example



Finished!

Prim‘s Algorithm

1. All vertices are marked as not visited

2. Any vertex ***v*** you like is chosen as starting vertex and is marked as visited (define a cluster ***C***)

3. The smallest- weighted edge ***e = (v,u)***, which connects one vertex *v* inside the cluster ***C*** with another vertex ***u*** outside of ***C***, is chosen and is added to the MST.

4. The process is repeated until a spanning tree is formed

Algorithm

- Prim(G=(N,A));graph; length):Set of Edges

  [Initialization]

  T= Empty Set

  B={An arbitrary member of N}

  while B != N do

    find e={u,v} of minimum length such that

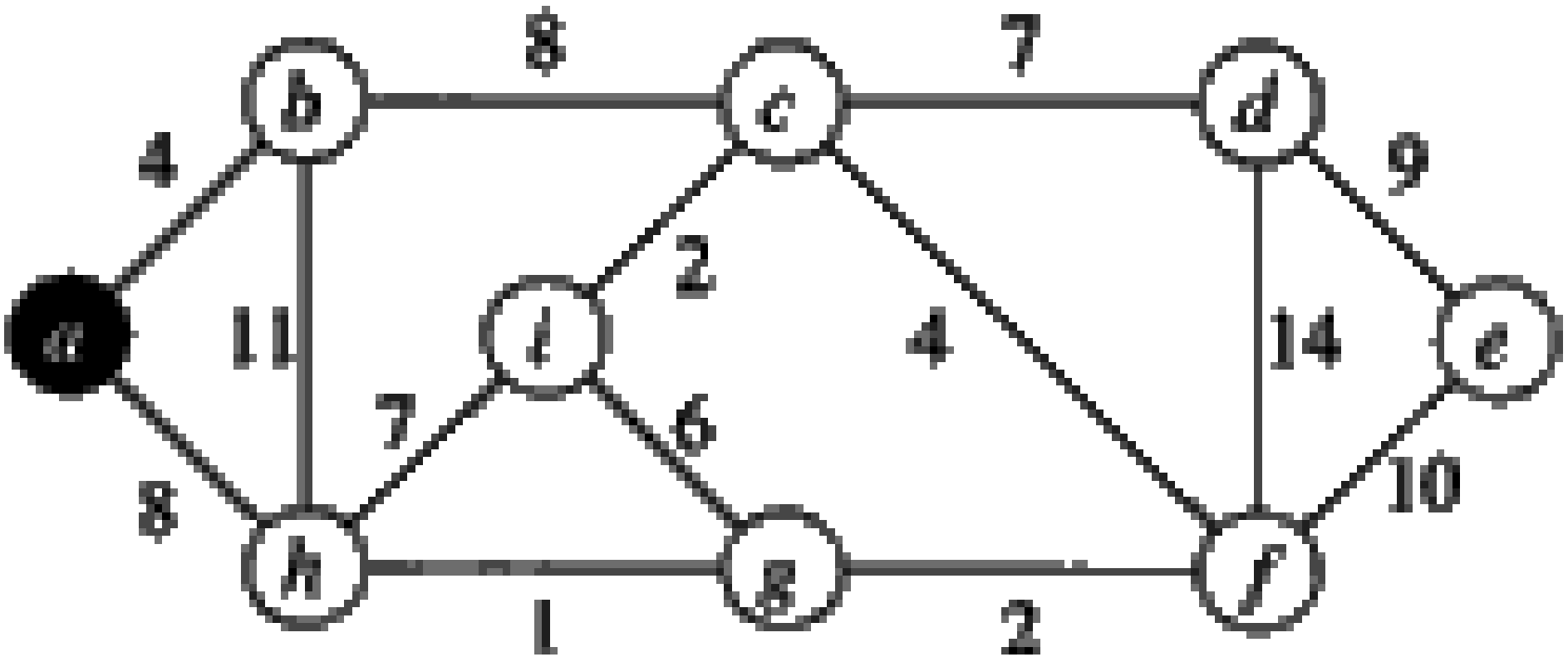
      u ∈ B and v ∈ N / B

      T = T U {e}

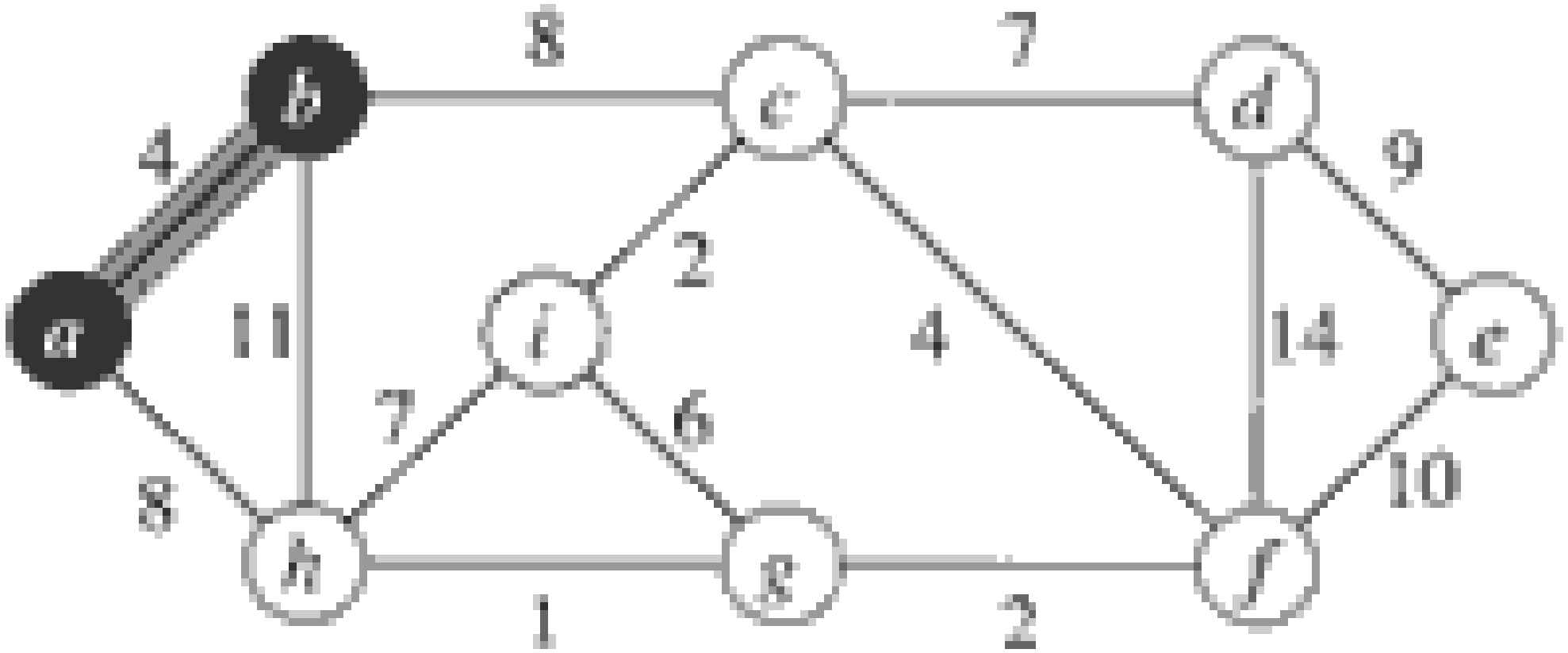
      B = B U {v}

  Return T

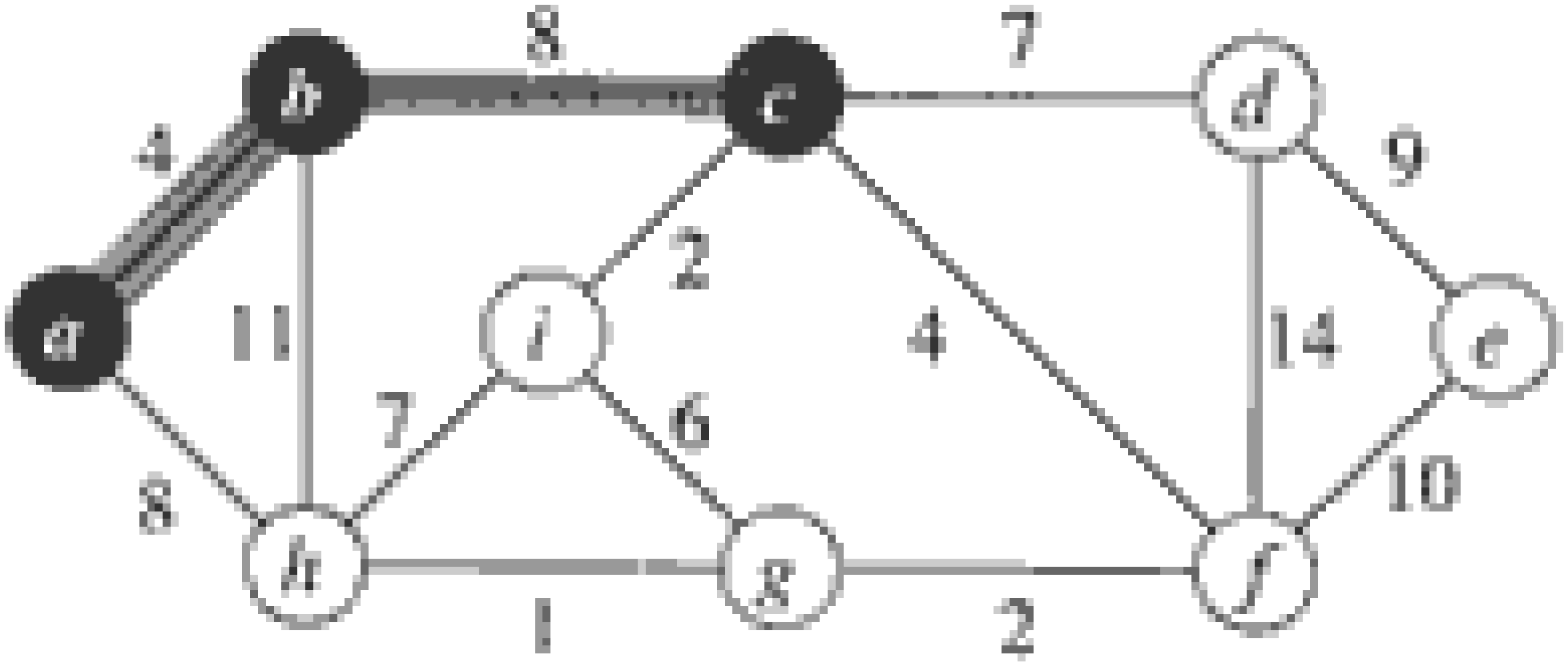
Prim's Algorithm: Example



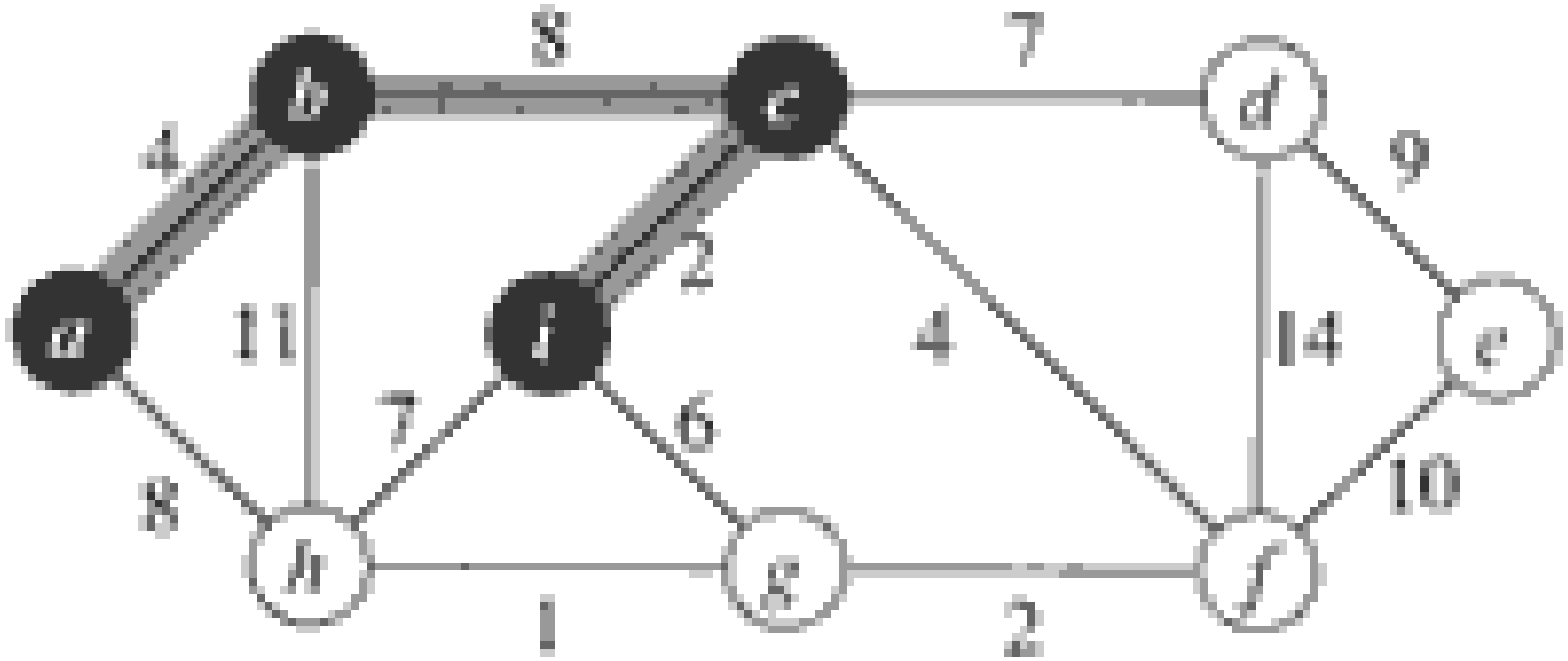
Prim's Algorithm: Example



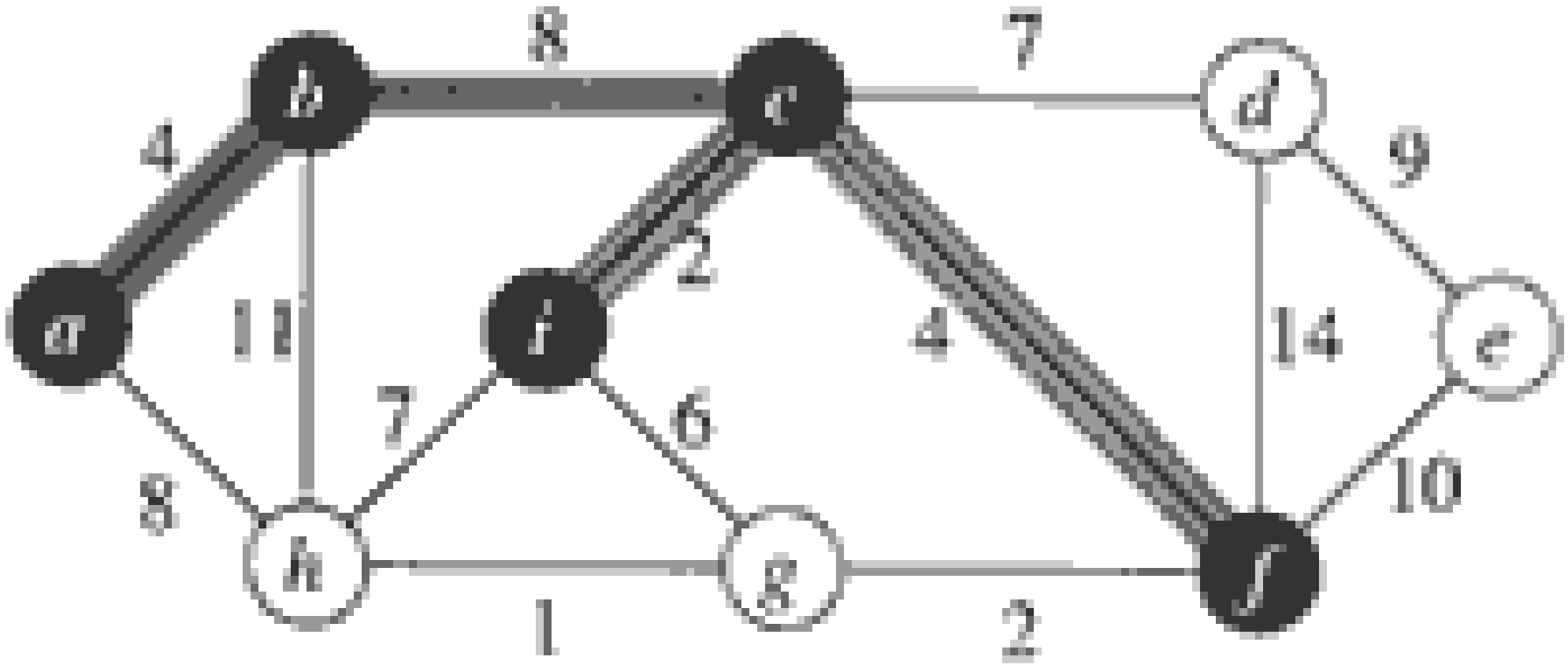
Prim's Algorithm: Example



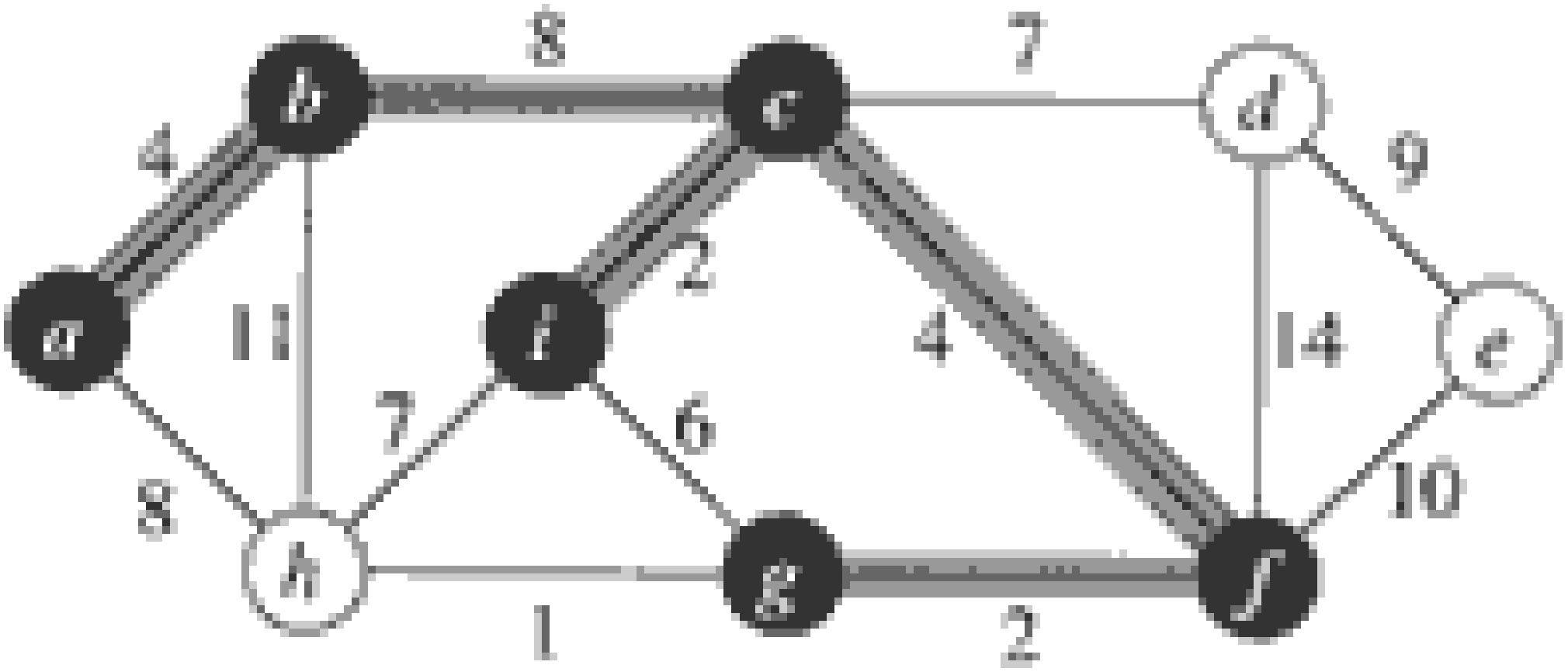
Prim's Algorithm: Example



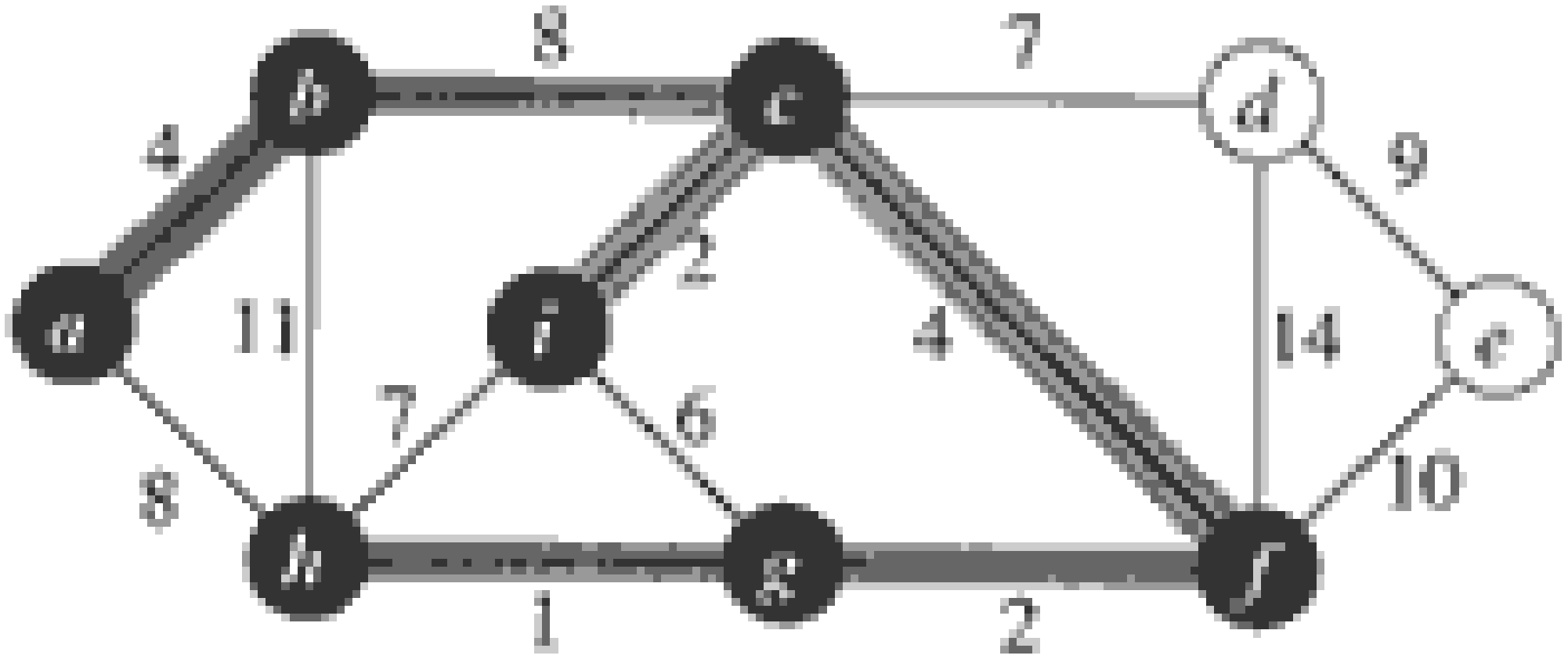
Prim's Algorithm: Example



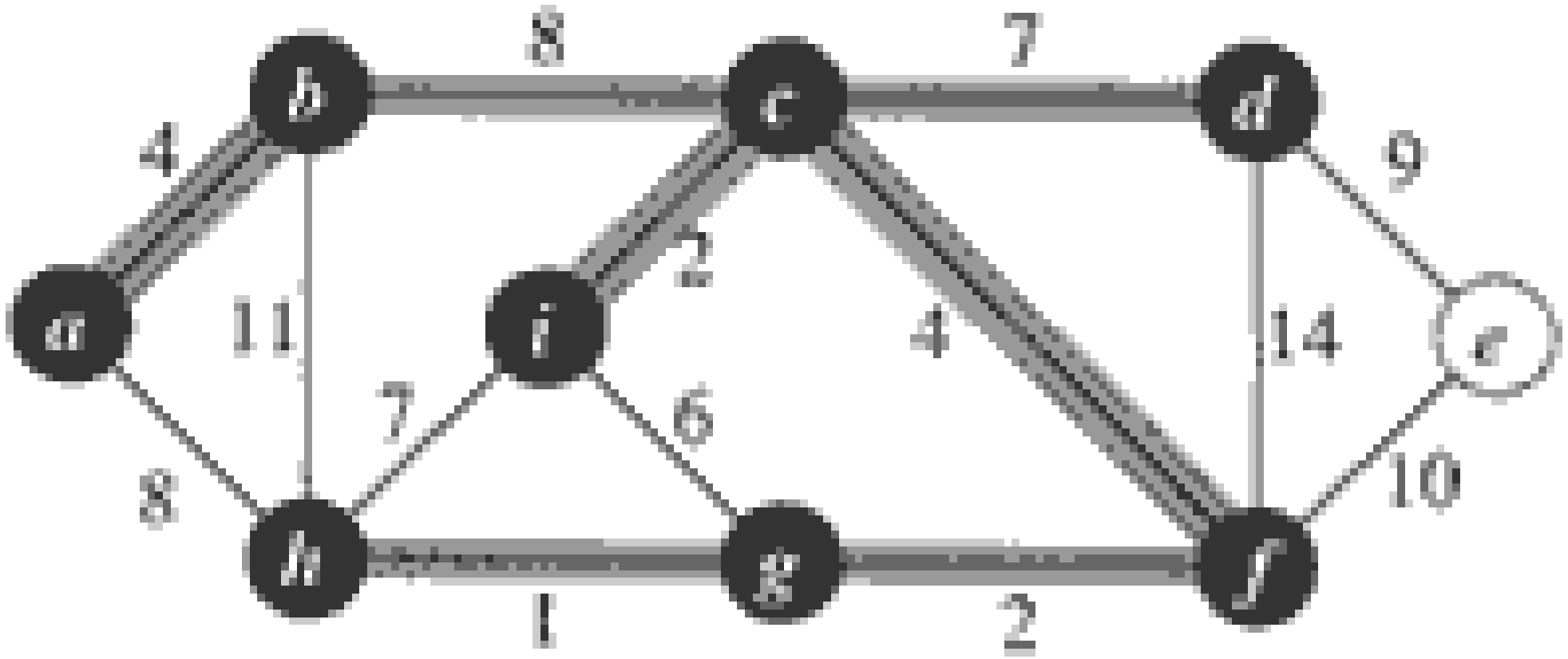
Prim's Algorithm: Example



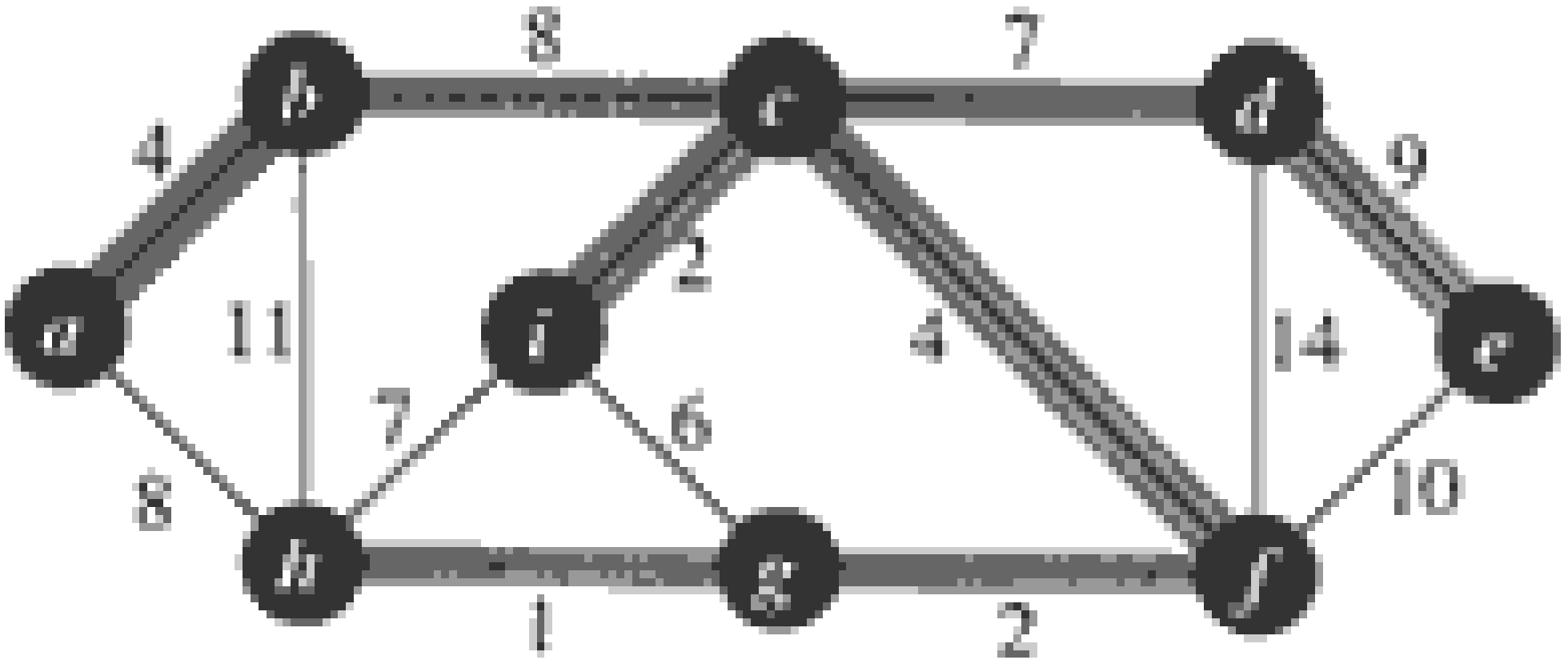
Prim's Algorithm: Example



Prim's Algorithm: Example



Prim's Algorithm: Example



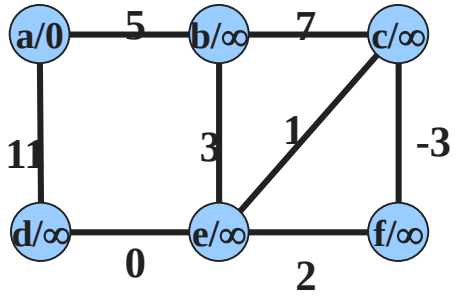
Finished!

Example

Step	{u,v}	B
Initialization	-	{a}
1	{a,b}	{a,b}
2	{b,c}	{a,b,c}
3	{c,i}	{a,b,c,i}
4	{c,f}	{a,b,c,i,f}
5	{f,g}	{a,b,c,i,f,g}
6	{g,h}	{a,b,c,I,f,g,h}
7	{c,d}	{a,b,c,d,I,f,g,h}
8	{d,e}	{a,b,c,d,e,f,g,h,i}

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```



Not in tree

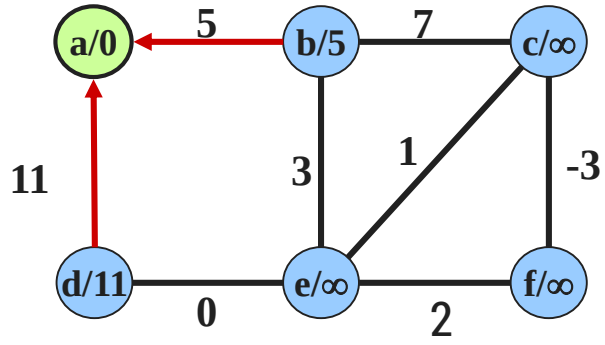
Q = a b c d e f

0 ∞ ⟷ ∞

⇦ decrease-key operation

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```

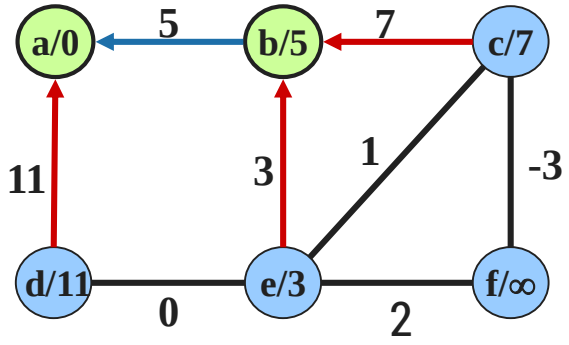


Q = b d c e f
5 11 ∞ ↔ ∞

⇦ decrease-key operation

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```

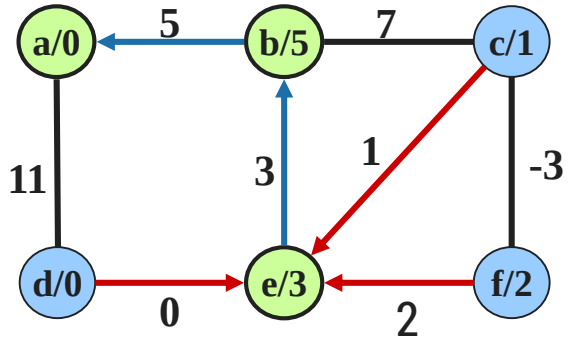


Q = e c d f
3 7 11 ∞

⇦ decrease-key operation

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```

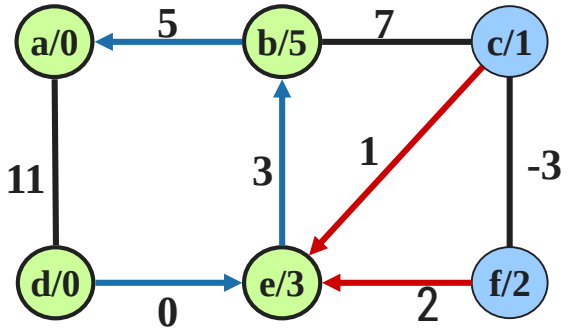


Q = d c f
0 1 2

⇦ decrease-key operation

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```

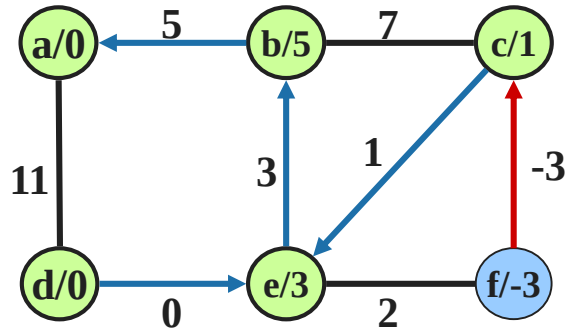


Q = c f
1 2

⇦ decrease-key operation

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)    ⇦ decrease-key operation
                fi
        od
od
```

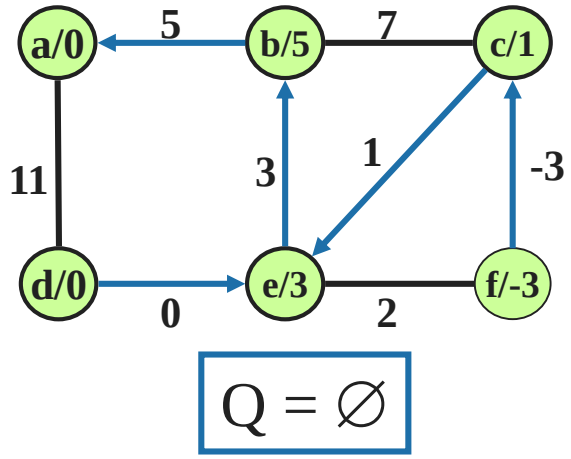


Q = f
-3

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```

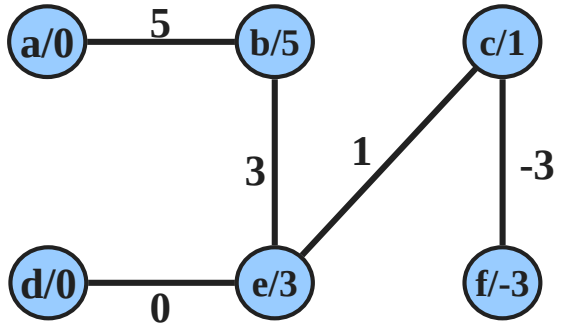
⇦ decrease-key operation



Prim's Algorithm

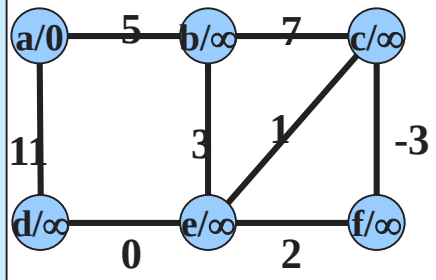
```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```

⇦ decrease-key operation



Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
        key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
        u := Extract - Min(Q);
        for each v ∈ Adj[u] do
                if v ∈ Q ∧ w(u, v) < key[v] then
                        π[v] := u;
                        key[v] := w(u, v)
                fi
        od
od
```



Not in tree

Q = a b c d e f

0 ∞ ∞

⇦ decrease-key operation

Prim's Algorithm

```
PRIM(V, E, w, r)
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
       π[u] ← NIL
       INSERT(Q, u)
DECREASE-KEY(Q, r, 0)      ▷ key[r] ← 0
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       for each v ∈ Adj[u]
           do if v ∈ Q and w(u, v) < key[v]
                 then π[v] ← u
                      DECREASE-KEY(Q, v, w(u, v))
```

O(V)

Executed |V| times

O(logV)

Executed |E| times

O(logV)

Running Time = $O(E \log V)$

Running Time = n^2

Algorithm Comparison

- Both Kruskal's and Prim's algorithm are greedy.
  - Kruskal's: Queue is static (constructed before loop)
  - Prim's: Queue is dynamic (keys adjusted as edges are encountered)
  - For Dense Graph edges tends to n(n-1)/2 and for Sparse graph edge tends towards n.
  - Kruskal : n^2 log n , n log n
  - Prims : n^2, n^2

Making Change Problem

Coin Change Problem

Solution: Greedy Approach.

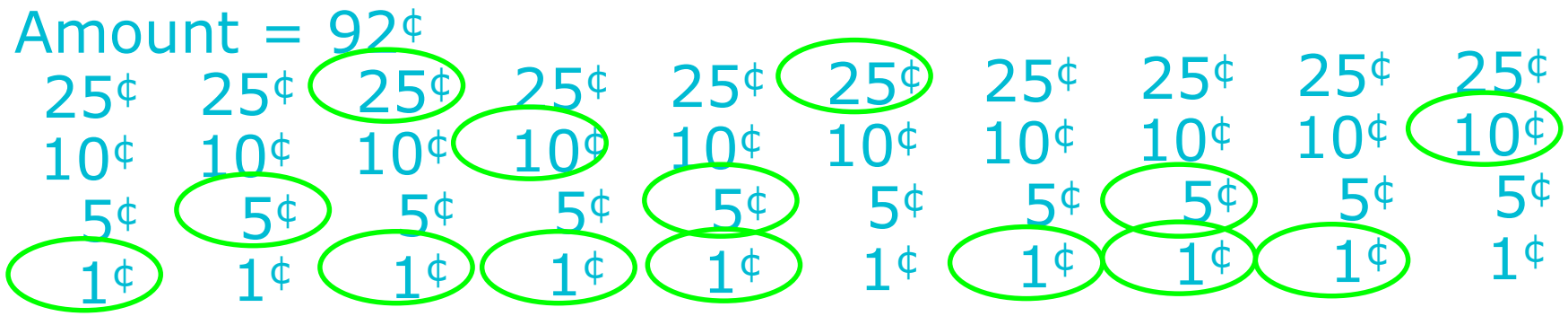
Approach: A common intuition would be to take coins with greater value first. This can reduce the total number of coins needed. Start from the largest possible denomination and keep adding denominations while the remaining value is greater than 0.

Algorithm:

1. Sort the array of coins in decreasing order.
2. Initialize result as empty.
3. Find the largest denomination that is smaller than current amount.
4. Add found denomination to result. Subtract value of found denomination from amount.
5. If amount becomes 0, then print result.
6. Else repeat steps 3 and 4 for new value of V.

Making Change Example

Instance: A drawer full of coins and an amount of change to return



Solutions for Instance: A subset of the coins that total the amount.

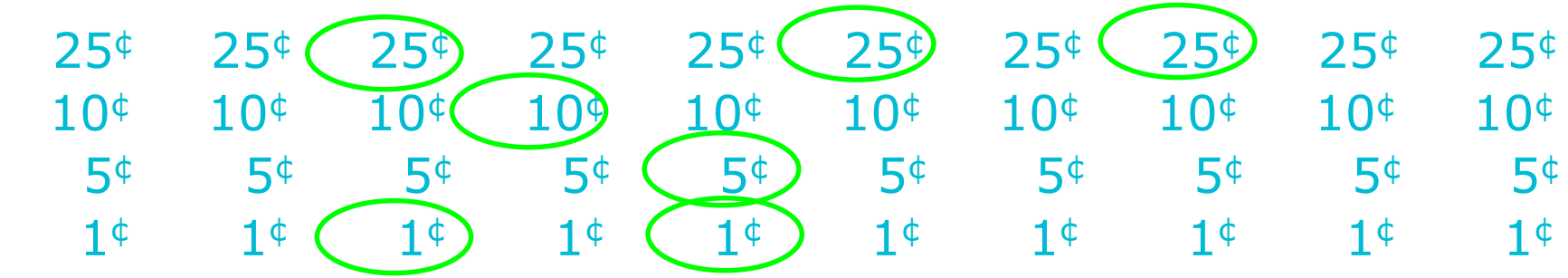
Cost of Solution: The number of coins in the solution = 14

Goal: Find an optimal valid solution.

Making Change Example

Instance: A drawer full of coins and an amount of change to return

Amount = 92¢



Greedy Choice:

Start by grabbing quarters until exceeds amount, then dimes, then nickels, then pennies.

Does this lead to an optimal # of coins?

Cost of Solution: 7

Formal Algorithm

- Make change for n units using the least possible number of coins.
- MAKE-CHANGE (n)

```
C ← {100, 25, 10, 5, 1} // constant.
  Sol ← {}; // set that will hold the solution set.
  Sum ← 0 sum of item in solution set
  WHILE sum not = n
        x = largest item in set C such that
            sum + x ≤ n
        IF no such item THEN
              RETURN "No Solution"
        S ← S {value of x}
        sum ← sum + x
  RETURN S
```

Hard Making Change Example

Problem: Find the minimum # of
4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4-cent coin.

Consequences:

4+1+1 = 6 mistake

3+3=6 better

Greedy Algorithm does not work!

Single Source shortest path problem

1) Dijkstra Algorithm
2) Bellman–Ford Algorithm

Dijkstra Algorithm-

•Dijkstra Algorithm is a very famous greedy algorithm.

•It is used for solving the single source shortest path problem.

•It computes the shortest path from one particular source node to all other remaining nodes of the graph.

Conditions-

It is important to note the following points regarding Dijkstra Algorithm-

•Dijkstra algorithm works only for connected graphs.

•Dijkstra algorithm works only for those graphs that do not contain any negative weight edge.

•The actual Dijkstra algorithm does not output the shortest paths.

•It only provides the value or cost of the shortest paths.

•By making minor modifications in the actual algorithm, the shortest paths can be easily obtained.

•Dijkstra algorithm works for directed as well as undirected graphs.

Algorithm

```
1.  dist[S] ← 0                          // The distance to source vertex is set to 0
2.  Π[S] ← NIL                           // The predecessor of source vertex is set as NIL
3.  for all v ∈ V - {S}                  // For all other vertices
4.        do dist[v] ← ∞                 // All other distances are set to ∞
5.        Π[v] ← NIL                     // The predecessor of all other vertices is set as NIL
6.  S ← ∅                                // The set of vertices that have been visited 'S' is initially empty
7.  Q ← V                                // The queue 'Q' initially contains all the vertices
8.  while Q ≠ ∅                          // While loop executes till the queue is not empty
9.        do u ← mindistance (Q, dist)   // A vertex from Q with the least distance is selected
10.         S ← S ∪ {u}                  // Vertex 'u' is added to 'S' list of vertices that have been
      visited
11.  for all v ∈ neighbors[u]            // For all the neighboring vertices of vertex 'u'
12.       do if dist[v] > dist[u] + w(u,v)   // if any new shortest path is discovered
13.             then dist[v] ← dist[u] + w(u,v)   // The new value of the shortest path is selected
14. return dist
```

Step-01:

In the first step. two sets are defined-
•One set contains all those vertices which have been included in the shortest path tree.
•In the beginning, this set is empty.
•Other set contains all those vertices which are still left to be included in the shortest path tree.
•In the beginning, this set contains all the vertices of the given graph.

Step-02:

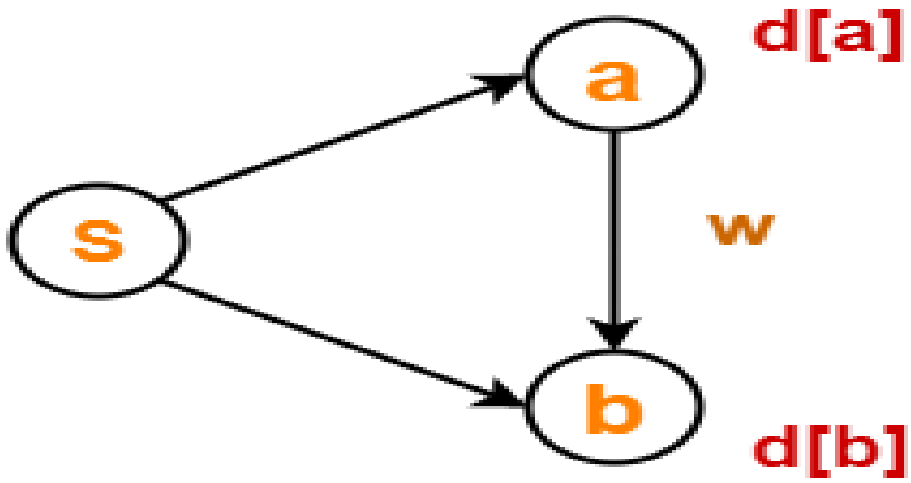
For each vertex of the given graph, two variables are defined as-
•Π[v] which denotes the predecessor of vertex 'v'
•d[v] which denotes the shortest path estimate of vertex 'v' from the source vertex.

Initially, the value of these variables is set as-
•The value of variable 'Π' for each vertex is set to NIL i.e. Π[v] = NIL
•The value of variable 'd' for source vertex is set to 0 i.e. d[S] = 0
•The value of variable 'd' for remaining vertices is set to ∞ i.e. d[v] = ∞

Step-03:

The following procedure is repeated until all the vertices of the graph are processed-

- Among unprocessed vertices, a vertex with minimum value of variable 'd' is chosen.
- Its outgoing edges are relaxed.
- After relaxing the edges for that vertex, the sets created in step-01 are updated.



Here, d[a] and d[b] denotes the shortest path estimate for vertices a and b respectively from the source vertex 'S'.

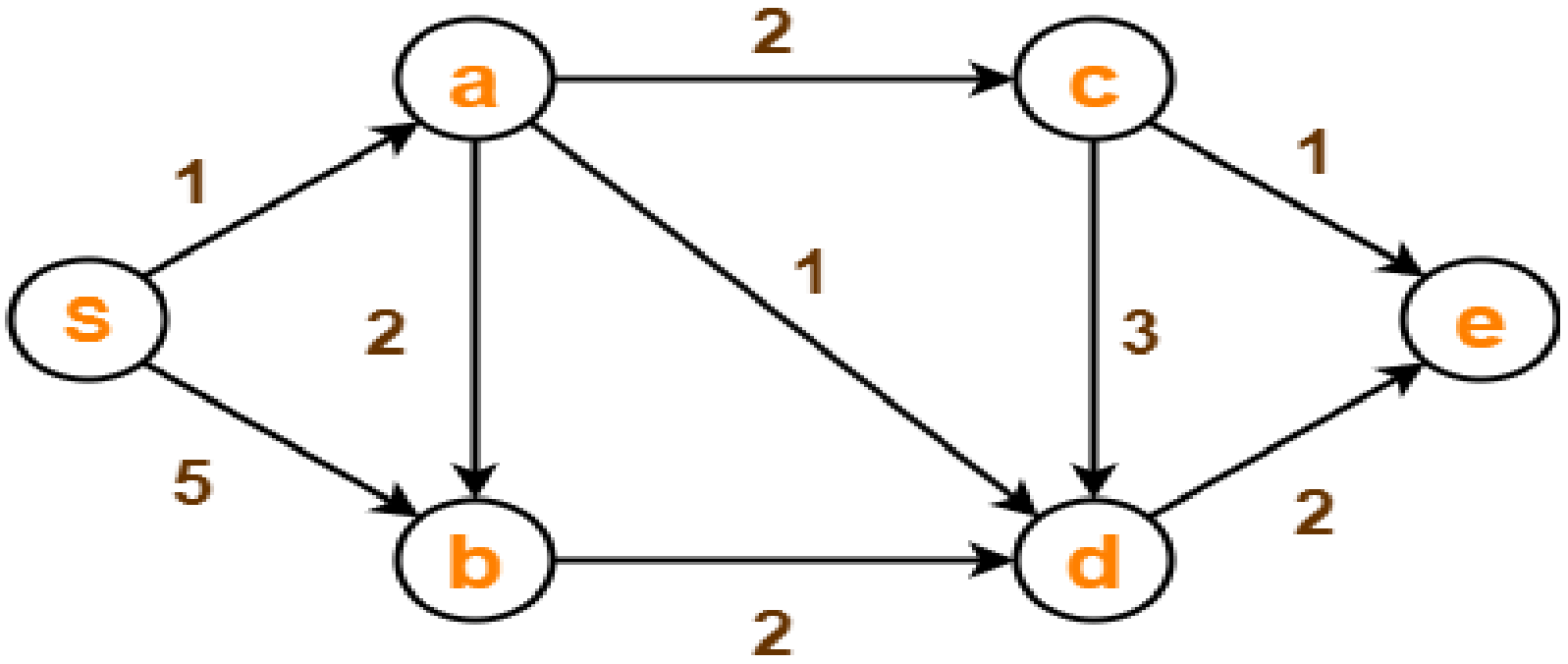
Now,

$$\text{If } d[a] + w < d[b]$$
$$\text{then } d[b] = d[a] + w \text{ and } \Pi[b] = a$$

This is called as edge relaxation.

Problem-

Using Dijkstra's Algorithm, find the shortest distance from source vertex 'S' to remaining vertices in the following graph- Also, write the order in which the vertices are visited.



Solution-

Step-01:

The following two sets are created-
•Unvisited set : {S , a , b , c , d , e}
•Visited set : { }

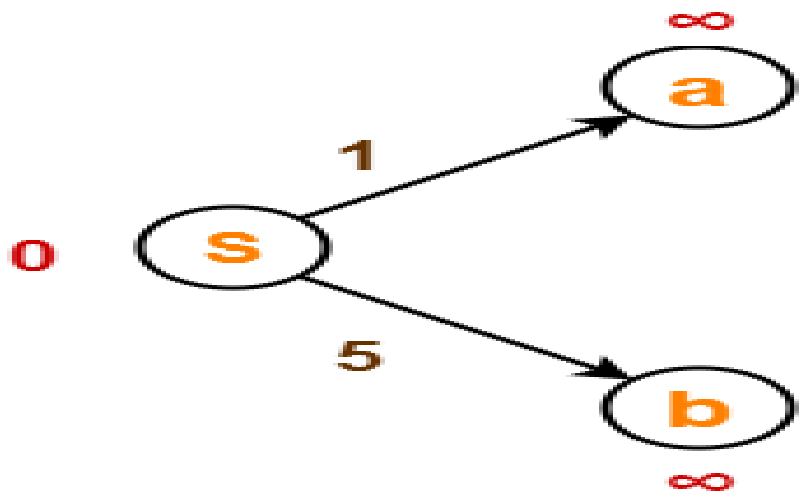
Step-02:

The two variables Π and d are created for each vertex and initialized as-
•$\Pi[S] = \Pi[a] = \Pi[b] = \Pi[c] = \Pi[d] = \Pi[e] = NIL$
•$d[S] = 0$
•$d[a] = d[b] = d[c] = d[d] = d[e] = \infty$

Step-03:

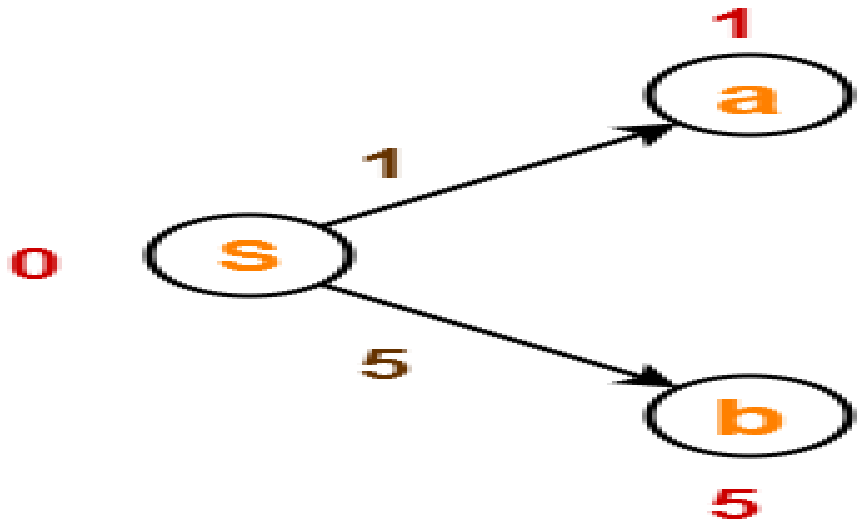
Vertex ‘S’ is chosen.
This is because shortest path estimate for vertex ‘S’ is least.
The outgoing edges of vertex ‘S’ are relaxed.

Before Edge Relaxation-



Now,
•d[S] + 1 = 0 + 1 = 1 < ∞
∴ d[a] = 1 and Π[a] = S
•d[S] + 5 = 0 + 5 = 5 < ∞
∴ d[b] = 5 and Π[b] = S

After edge relaxation, our shortest path tree is

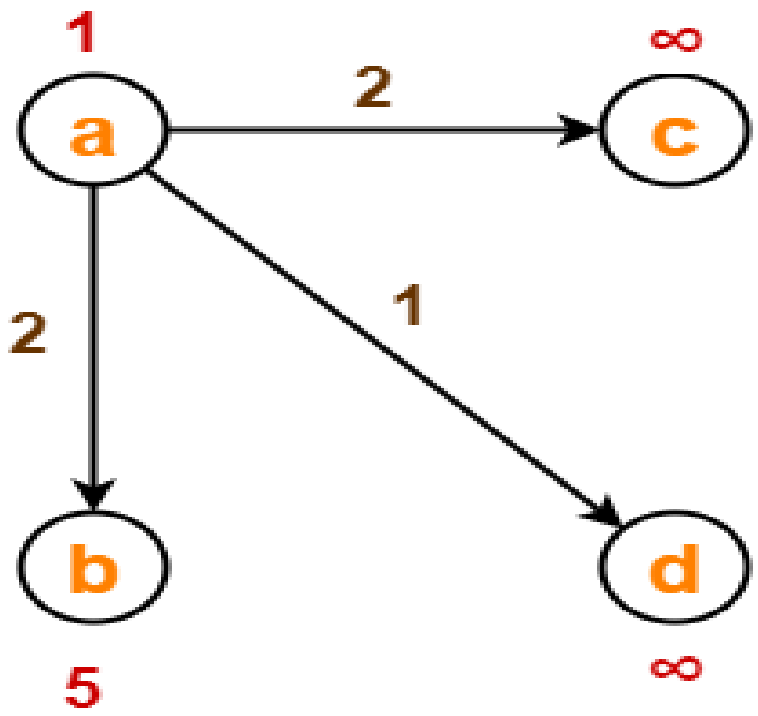


Now, the sets are updated as-
•Unvisited set : {a , b , c , d , e}
•Visited set : {S}

Step-04:

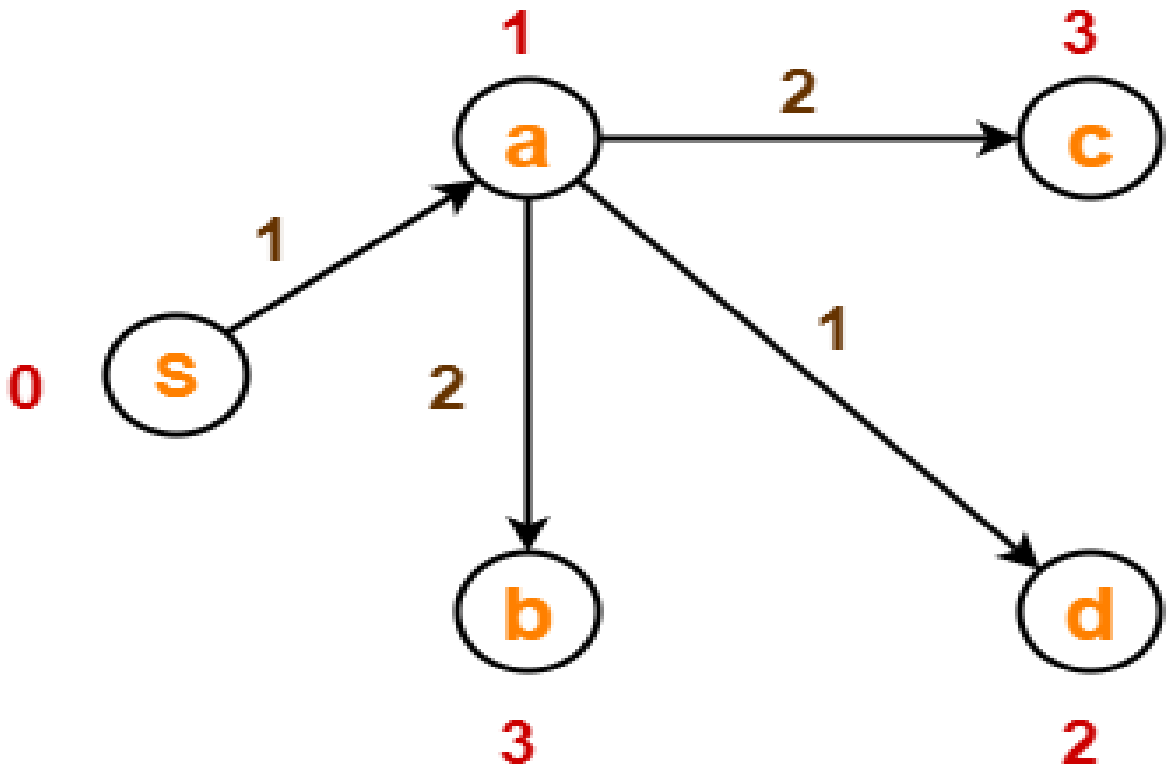
•Vertex ‘a’ is chosen.
•This is because shortest path estimate for vertex ‘a’ is least.
•The outgoing edges of vertex ‘a’ are relaxed.

Before Edge Relaxation-



Now,
•d[a] + 2 = 1 + 2 = 3 < ∞
∴ d[c] = 3 and Π[c] = a
•d[a] + 1 = 1 + 1 = 2 < ∞
∴ d[d] = 2 and Π[d] = a
•d[b] + 2 = 1 + 2 = 3 < 5
∴ d[b] = 3 and Π[b] = a

After edge relaxation, our shortest path tree is-

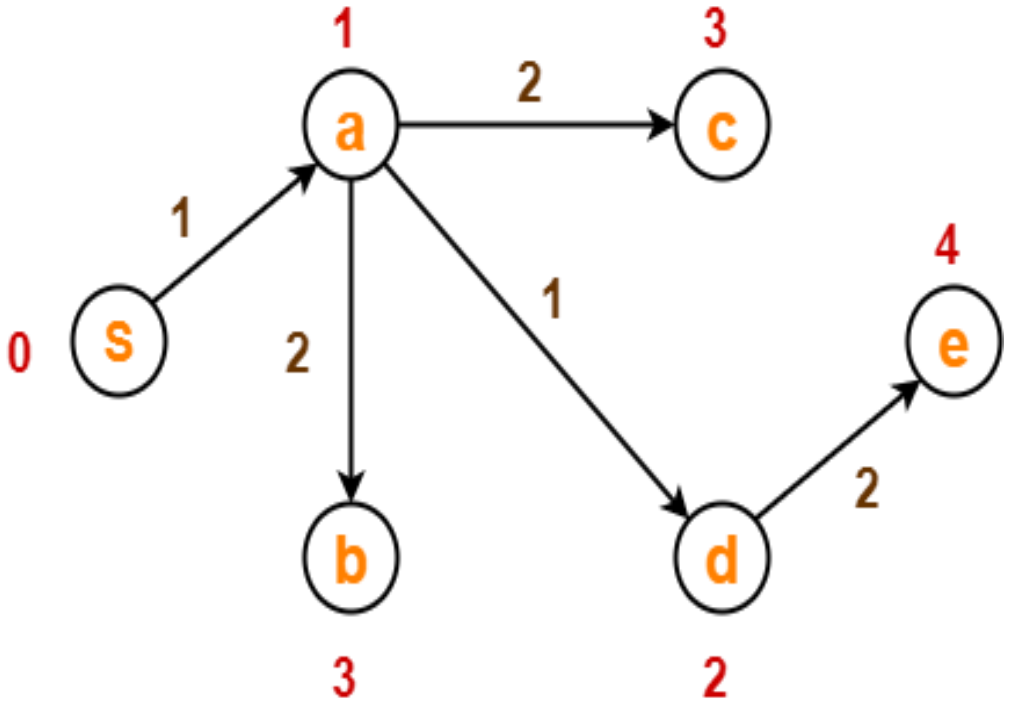


Now, the sets are updated as-

- Unvisited set : {b , c , d , e}
- Visited set : {S , a}

Step-05:

•Vertex ‘d’ is chosen.
•This is because shortest path estimate for vertex ‘d’ is least.
•The outgoing edges of vertex ‘d’ are relaxed.



Now,
•d[d] + 2 = 2 + 2 = 4 < ∞
∴ d[e] = 4 and Π[e] = d

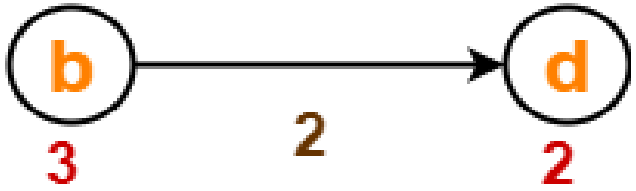
After edge relaxation, our shortest path tree is-

Now, the sets are updated as-
•Unvisited set : {b , c , e}
•Visited set : {S , a , d}

Step-06:

•Vertex ‘b’ is chosen.
•This is because shortest path estimate for vertex ‘b’ is least.
•Vertex ‘c’ may also be chosen since for both the vertices, shortest path estimate is least.
•The outgoing edges of vertex ‘b’ are relaxed.

Before Edge Relaxation-



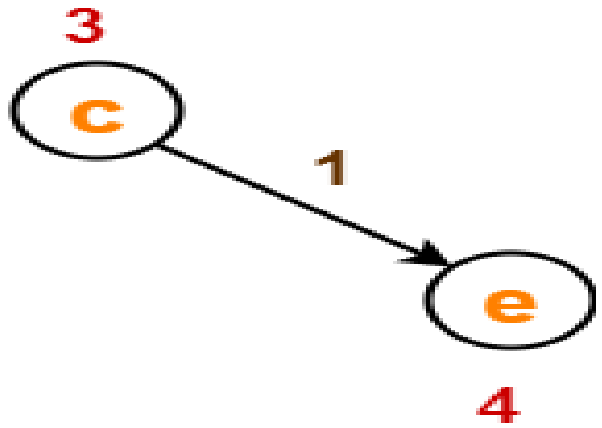
Now,
•d[b] + 2 = 3 + 2 = 5 > 2
∴ No change

After edge relaxation, our shortest path tree remains the same as in Step-05.
Now, the sets are updated as-
•Unvisited set : {c , e}
•Visited set : {S , a , d , b}

Step-07:

•Vertex ‘c’ is chosen.
•This is because shortest path estimate for vertex ‘c’ is least.
•The outgoing edges of vertex ‘c’ are relaxed.

Before Edge Relaxation-



Now,
•d[c] + 1 = 3 + 1 = 4 = 4
∴ No change

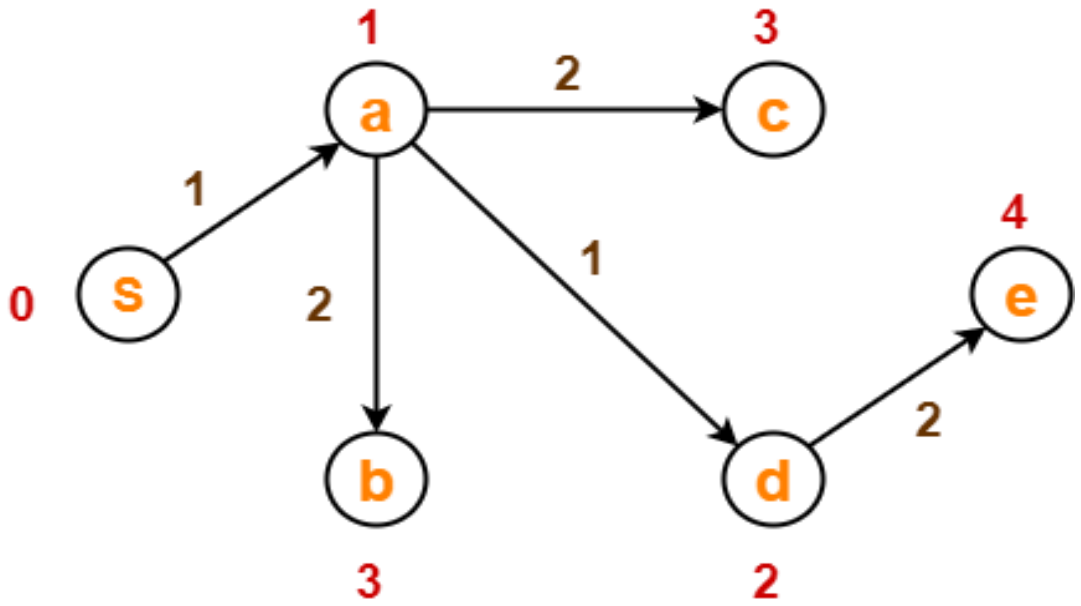
After edge relaxation, our shortest path tree remains the same as in Step-05.
Now, the sets are updated as-
•Unvisited set : {e}
•Visited set : {S , a , d , b , c}

Step-08:

•Vertex ‘e’ is chosen.
•This is because shortest path estimate for vertex ‘e’ is least.
•The outgoing edges of vertex ‘e’ are relaxed.
•There are no outgoing edges for vertex ‘e’.
•So, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-
•Unvisited set : { }
•Visited set : {S , a , d , b , c , e}

Now,
•All **vertices** of the graph are processed.
•Our final shortest path tree is as shown below.
•It represents the shortest path from source vertex ‘S’ to all other remaining vertices.



Shortest Path Tree

The order in which all the vertices are processed is :

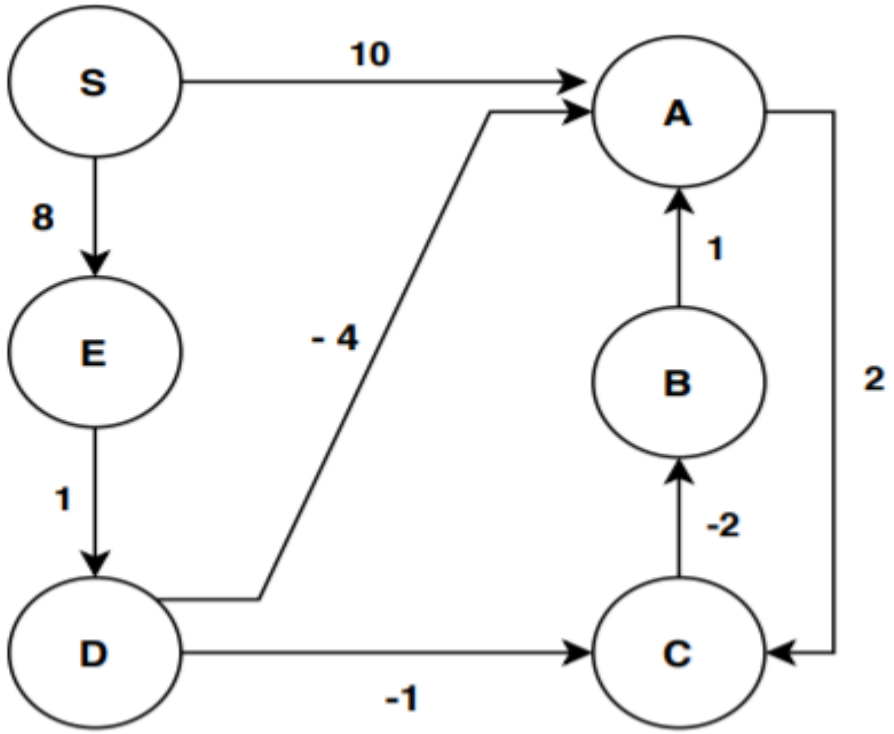
S , a , d , b , c , e.

Bellman–Ford Algorithm

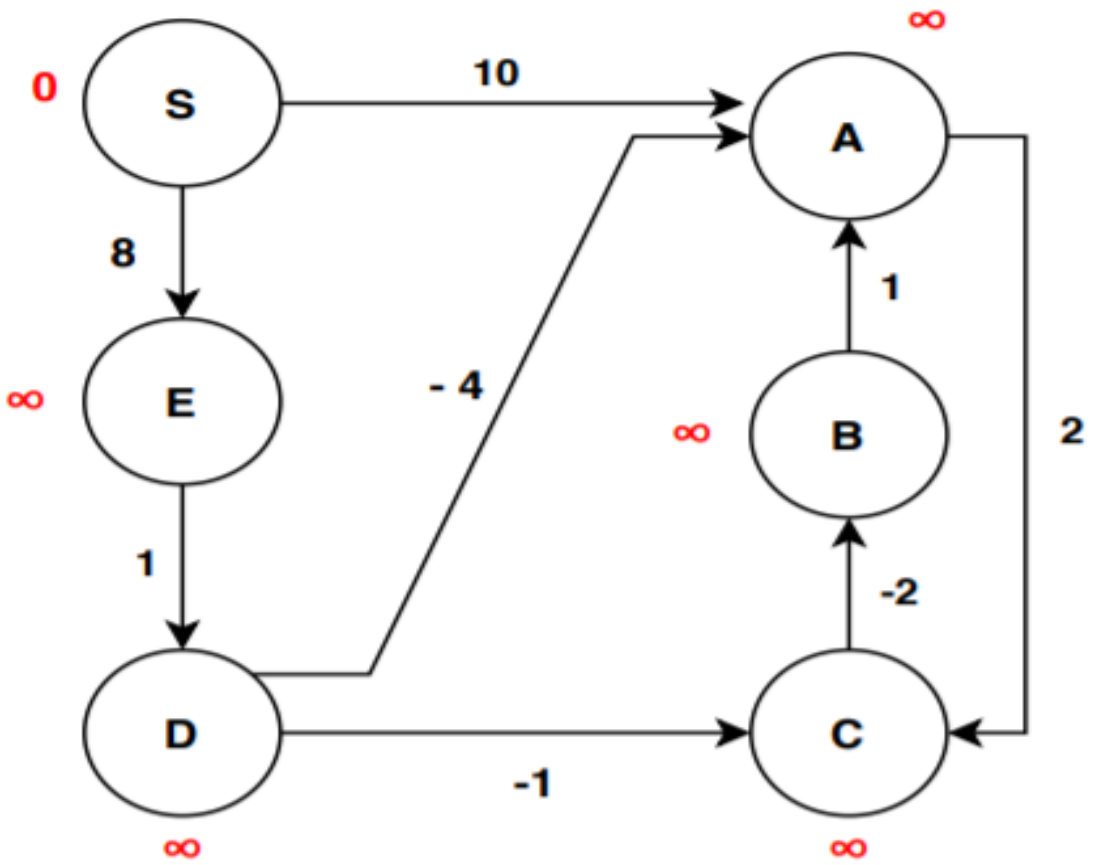
We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and the time complexity is O((V+E)LogV) .

Dijkstra doesn't work for Graphs with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(V * E), which is more than Dijkstra.

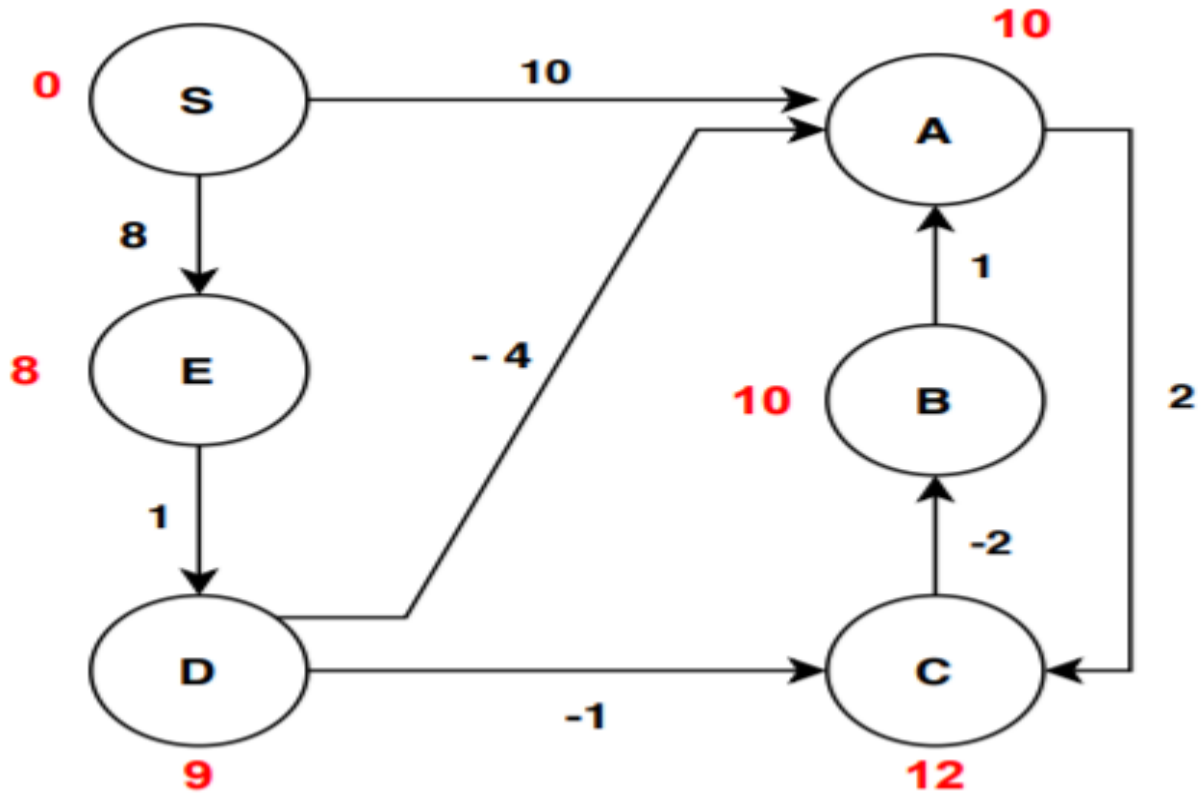
Example - A Graph Without Negative Cycle



Assume that S is our starting vertex. We're now ready to start with the initialization step of the algorithm:

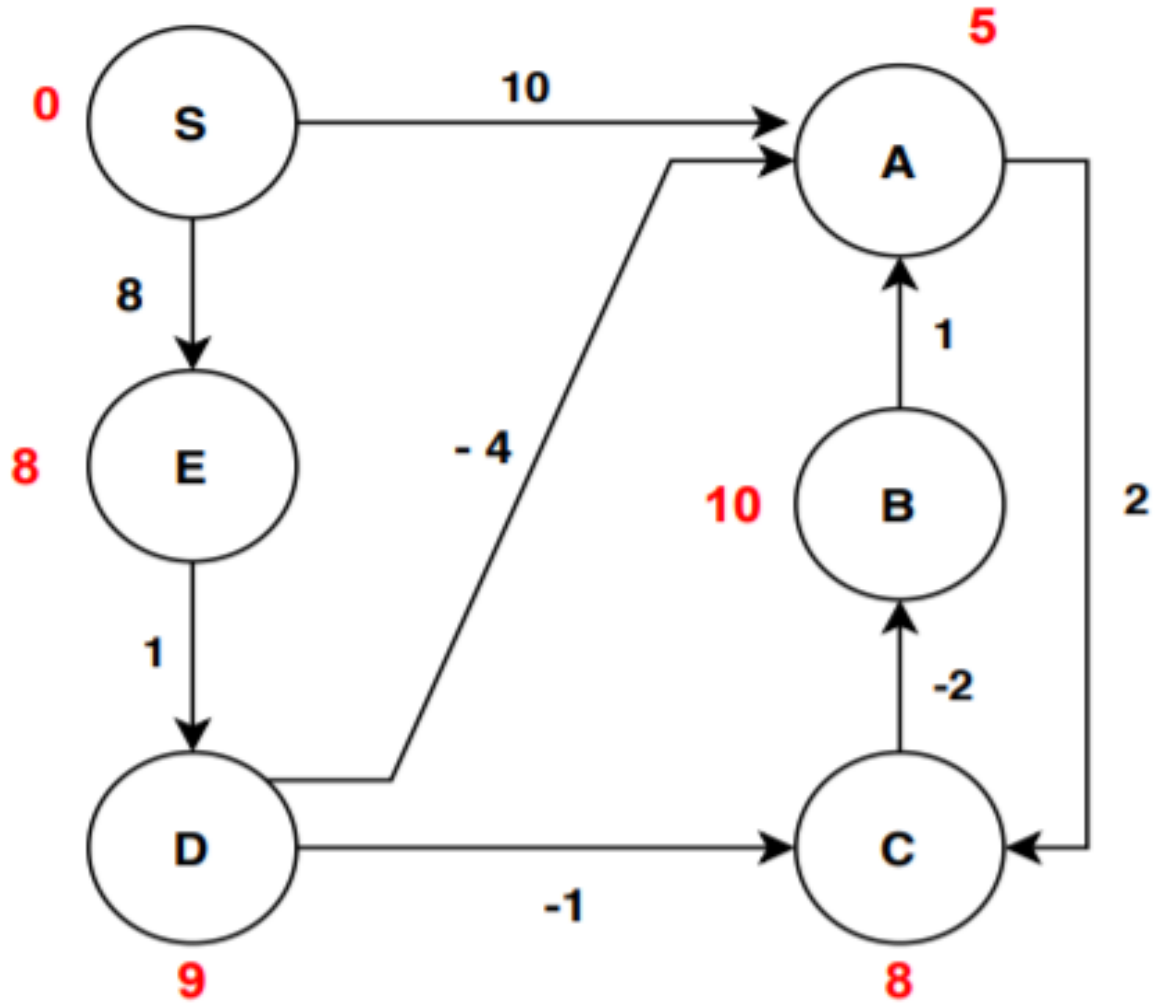


After initialization the graph, we can now proceed to the **first iteration:**

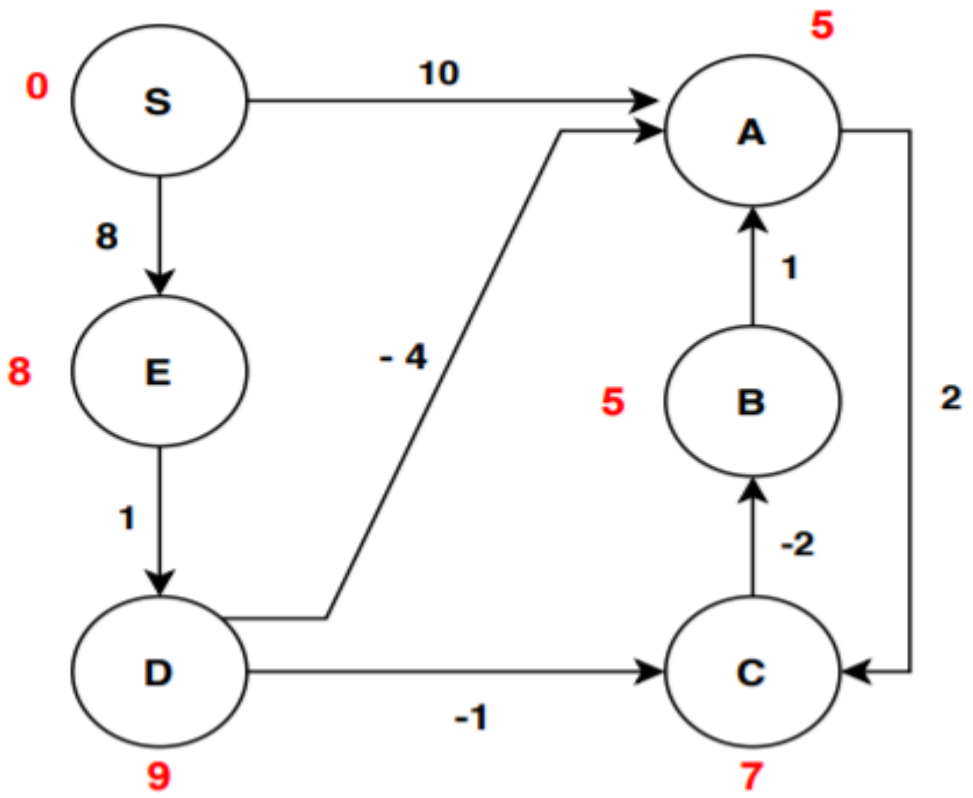


Second iteration:

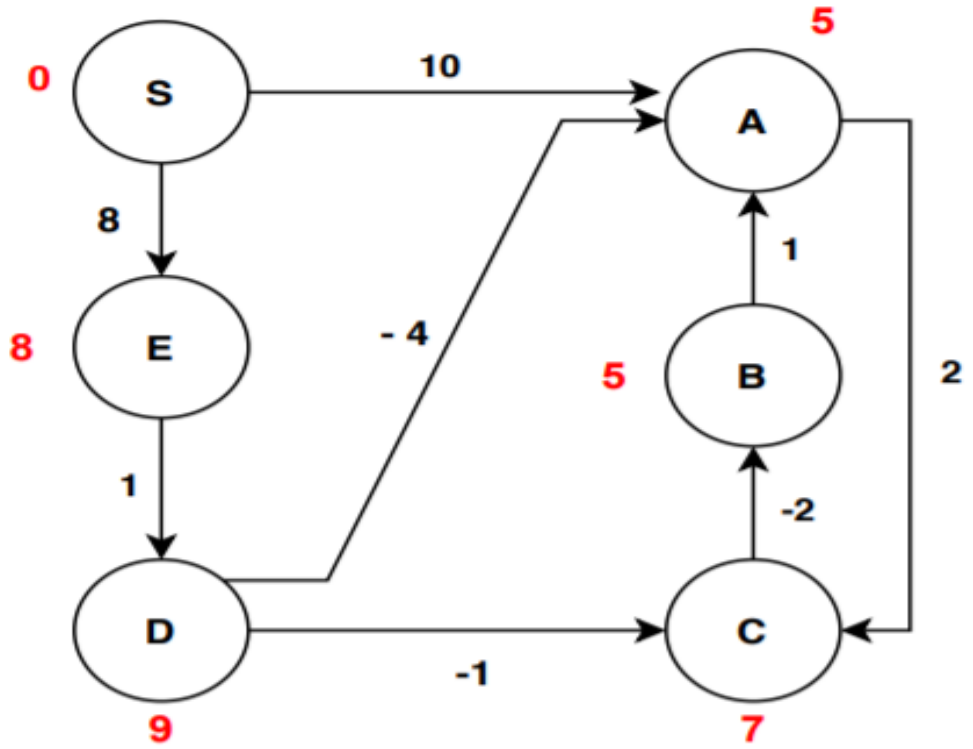
(S, A) -> (S, E) -> (A, C) -> (B, A) -> (C, B) -> (D, C) -> (D, A) -> (E, D)
Now we're done with the first iteration. Let's see how the graph changes after the second iteration:



Third iteration:

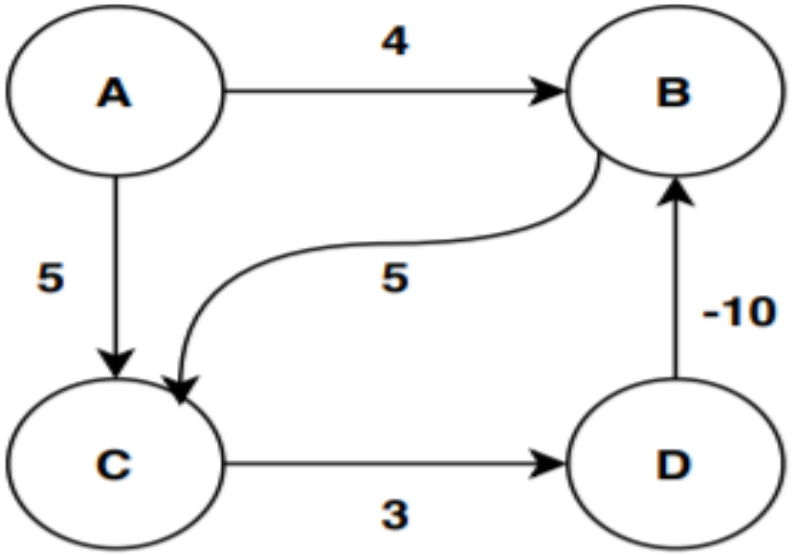


Fourth iteration



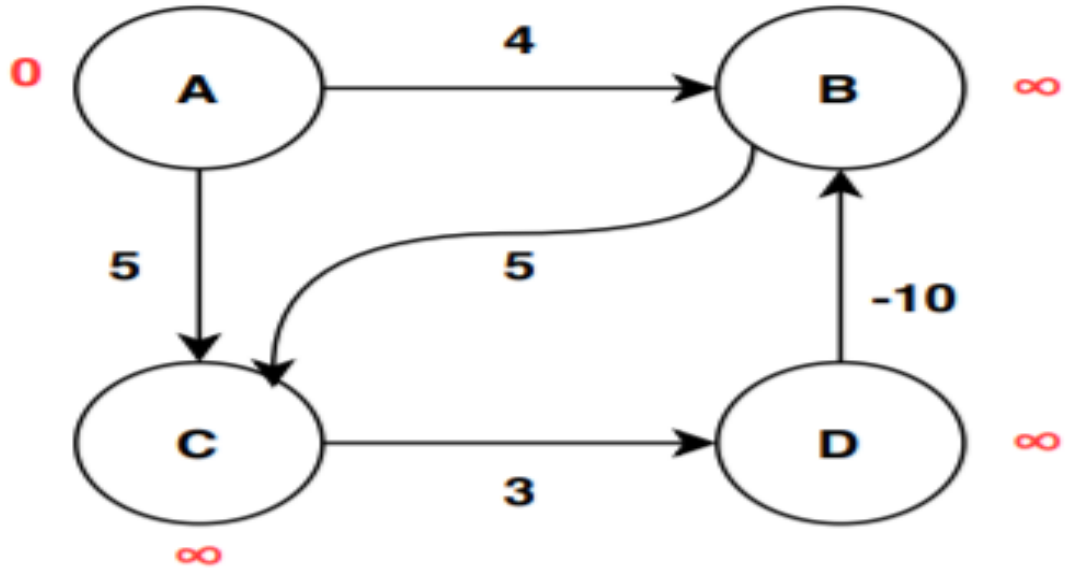
There are no updates in the distance values of vertices after the fourth iteration. This means the algorithms produce the final result. Now, we mentioned that we need to run this algorithm for 5 interactions. But in this case, we got our result after 4.
In this case, we got the same values for two consecutive iterations hence the algorithm terminates.

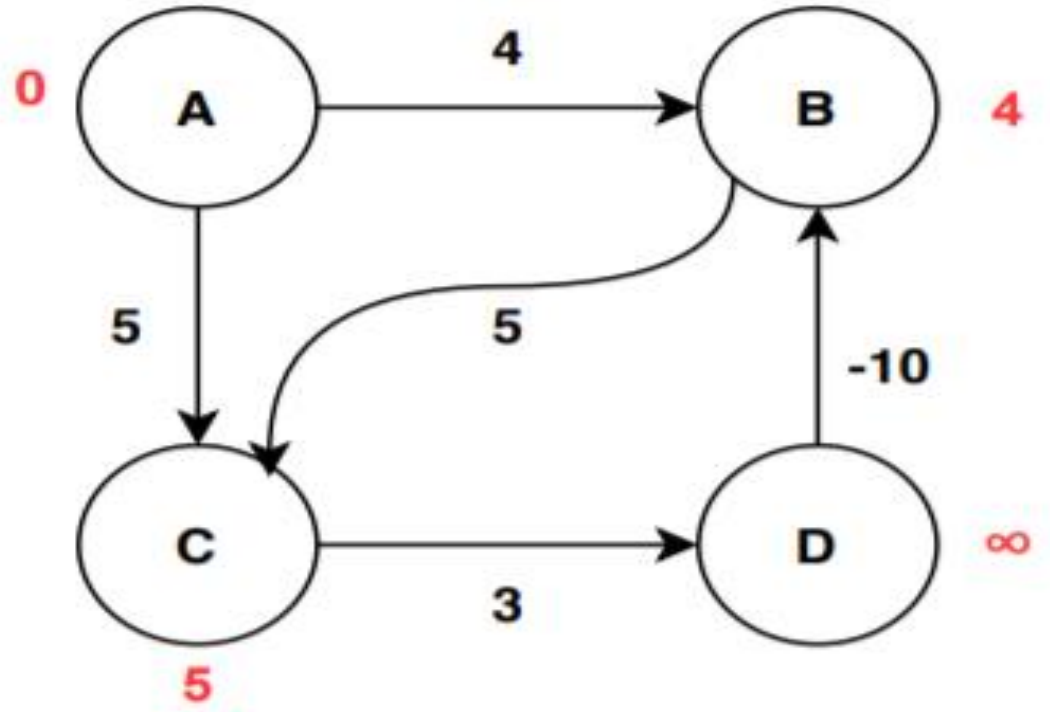
Example- A Graph With Negative Cycle



The graph has 4 vertices. We're considering the vertex A as the starting vertex here. The algorithm expects to iterate 3 times for calculation of shortest distance and one more time to check for the negative cycle.

The first step is to initialize the graph:

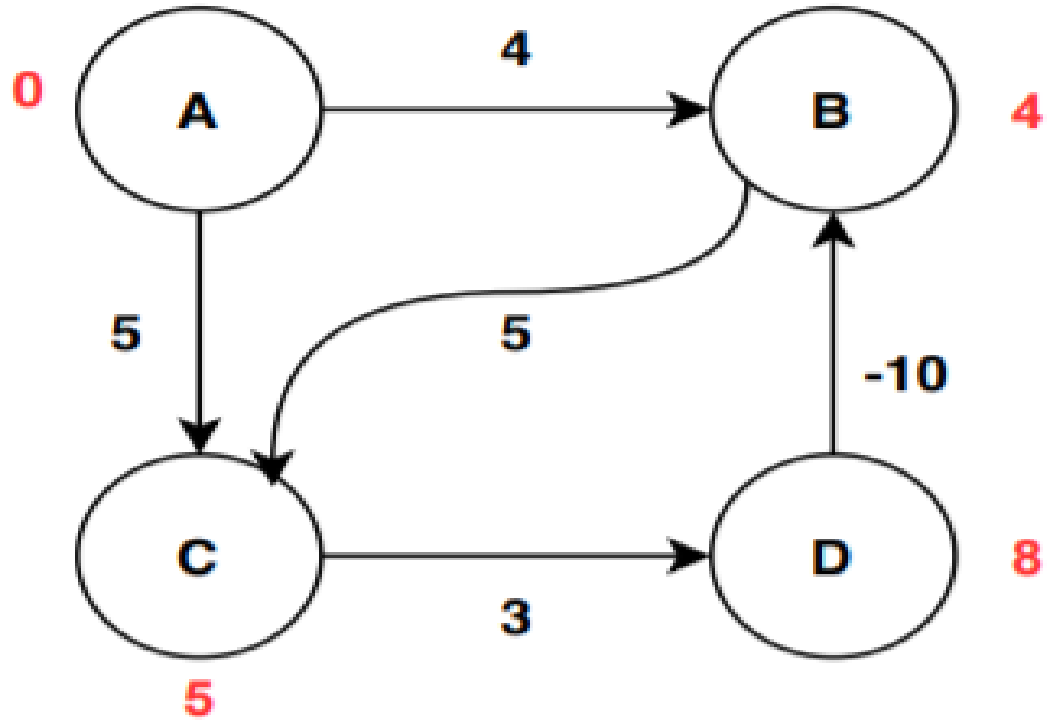




After the first iteration, we can see changes in the distance value of B and C. From the vertex A, the edge weights are directly assigned to vertex B and C to update their distance value. For edge (A, C):

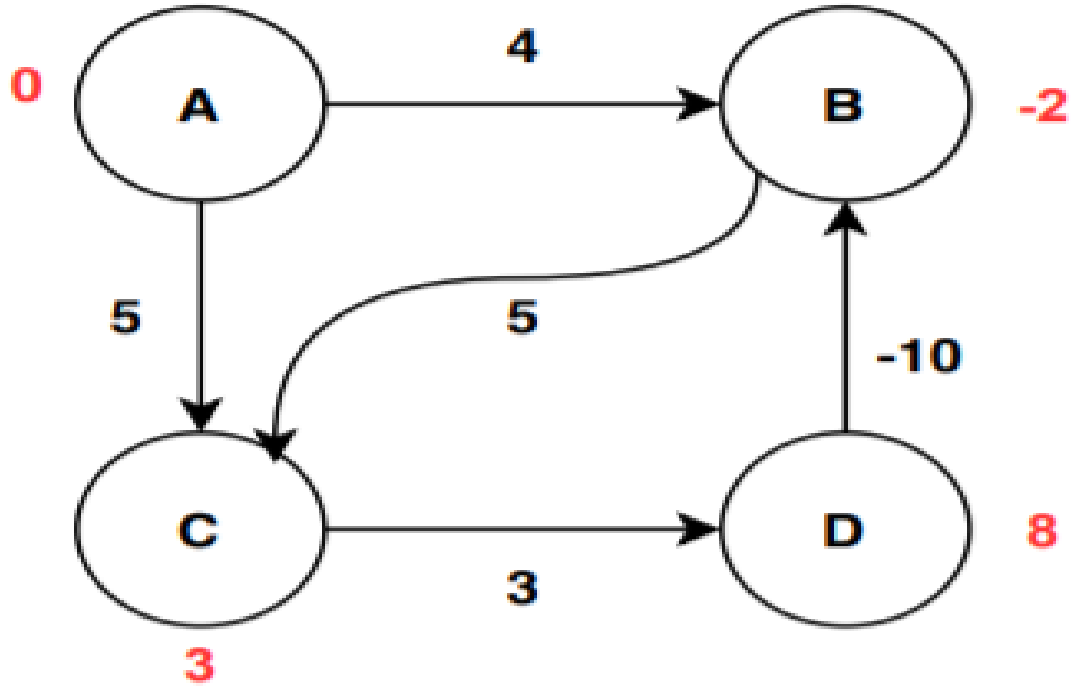
Therefore, the new value of the vertex C is updated:

Similarly for edge (A, B):



After the second iteration, the distance value of the vertex D is updated by the algorithm by relaxing the edge (C, D):

Therefore, the new value of the vertex C is updated:



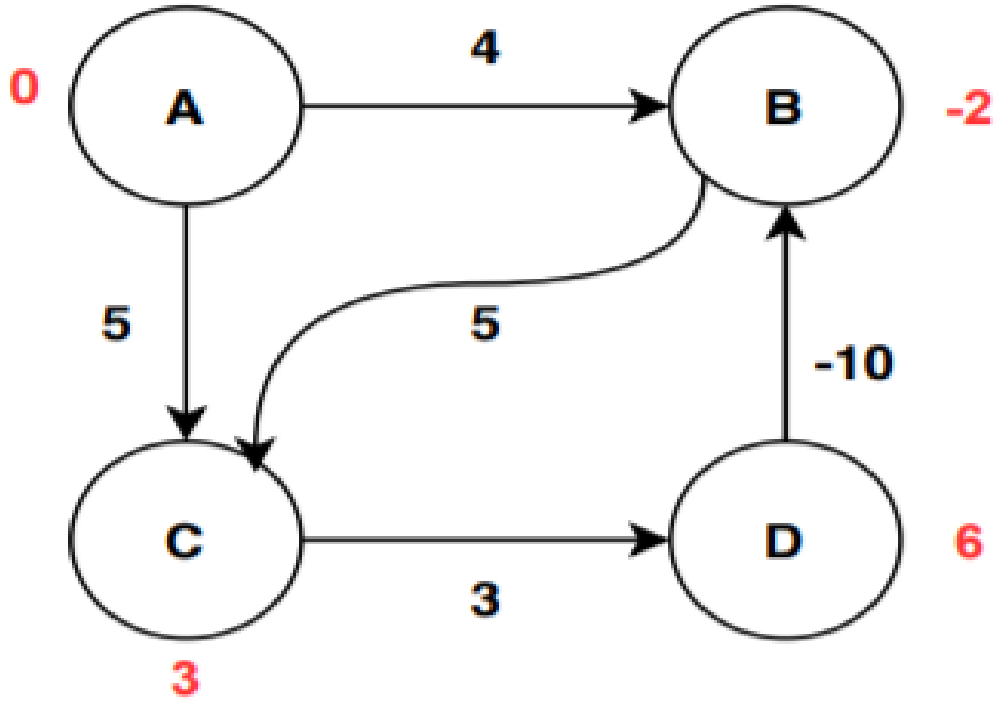
Here there are two updates. One for the vertex B and another for vertex C. The update for vertex B is done by relaxing the edge (D, B):

Therefore, the new value of the vertex B is updated:

The algorithm updated the value for vertex B by relaxing the edge (B, C):

The value of C is updated and stored:

We're done with the maximum required iterations. Now let's iterate one last time to decide whether the graph has a negative cycle or not:



The distance values are not stable even after the maximum number of iterations. Therefore, in this case, the algorithms return that the graph contains a negative weighted cycle, and hence it is not possible to calculate the shortest path from the starting vertex to all other vertices in the given graph.

Time Complexity Analysis

First, the initialization step takes $O(V)$.

Then, the algorithm iterates $(|V| - 1)$ times with each iteration taking $O(1)$ time.

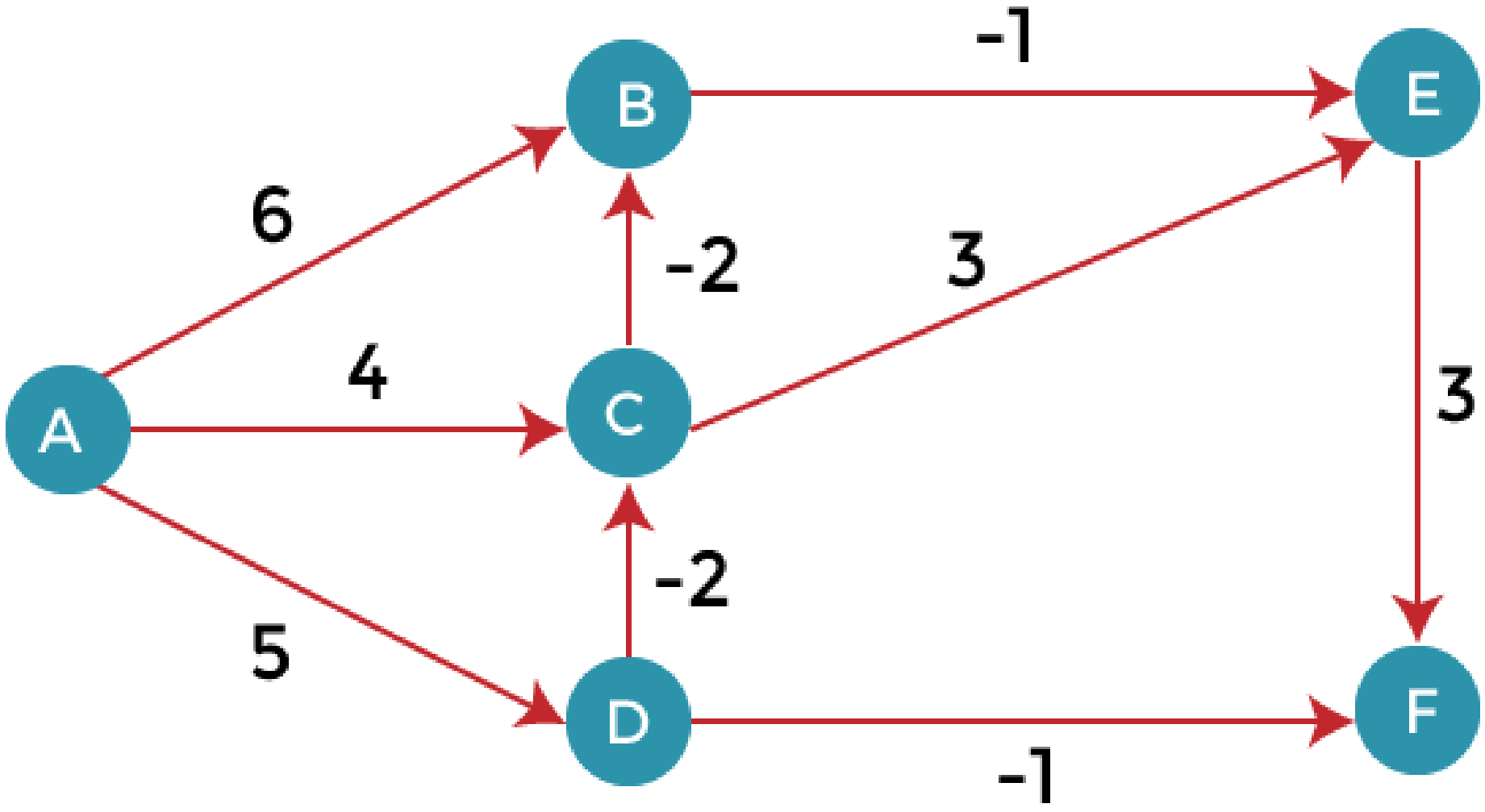
After $(|V| - 1)$ interactions, the algorithm chooses all the edges and then passes the edges to *Relax()*. Choosing all the edges takes $O(E)$ time and the function *Relax()* takes $O(1)$ time.

Therefore **the complexity to do all the operations takes $O(VE)$ time.**

Within the *Relax()* function, the algorithm takes a pair of edges, does a checking step, and assigns the new weight if satisfied. All these operations take $O(E)$ times.

Thus the total time of the Bellman-Ford algorithm is the sum of initialization time, *for* loop time, and *Relax function* time. **In total, the time complexity of the Bellman-Ford algorithm is $O(VE)$.**

Practice Example



Huffman Coding

Huffman Coding

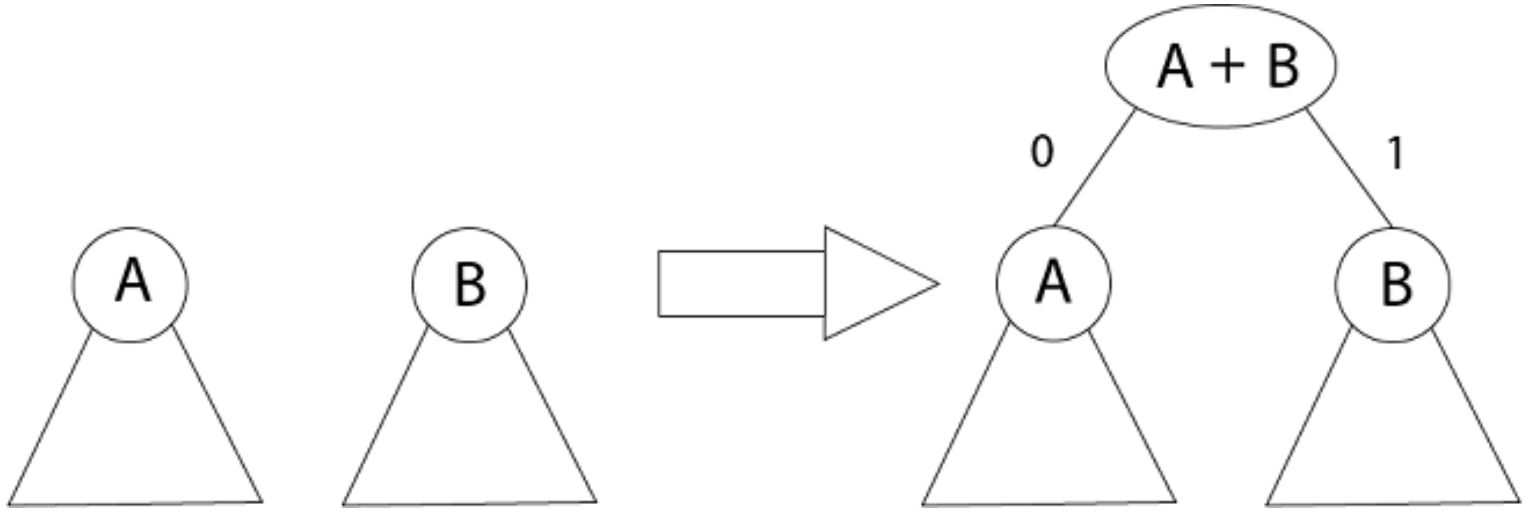
Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

(i) Data can be encoded efficiently using Huffman Codes.
(ii) It is a widely used and beneficial technique for compressing data.
(iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Greedy Algorithm for constructing a Huffman Code

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.



The algorithm builds the tree T analogous to the optimal code in a bottom-up manner. It starts with a set of |C| leaves (C is the number of characters) and performs |C| - 1 'merging' operations to create the final tree.

Algorithm of Huffman Code

Huffman (C)

1. n=|C|
2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x= left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)

Huffman Coding

There are mainly two major parts in Huffman Coding

(1) Build a Huffman Tree from input characters.

(2) Traverse the Huffman Tree and assign codes to characters.

Example: Find an optimal Huffman Code for the following set of frequencies:

a: 50 b: 25 c: 15 d: 40 e: 75

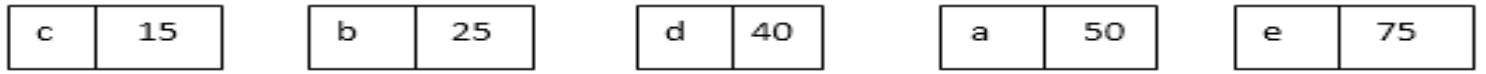
Solution: Given that: C = {a, b, c, d, e}

f (C) = {50, 25, 15, 40, 75}

n = 5

Q ← c

i.e.

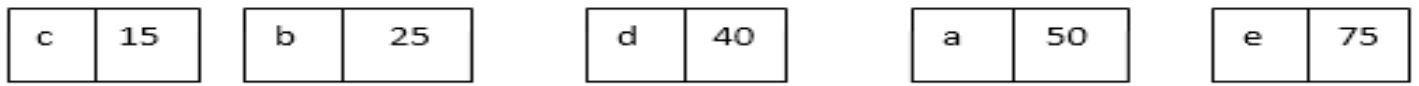


for i ← 1 to 4

i = 1 Z ← Allocate node

x ← Extract-Min (Q)

y ← Extract-Min (Q)

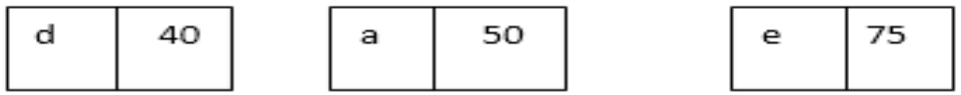


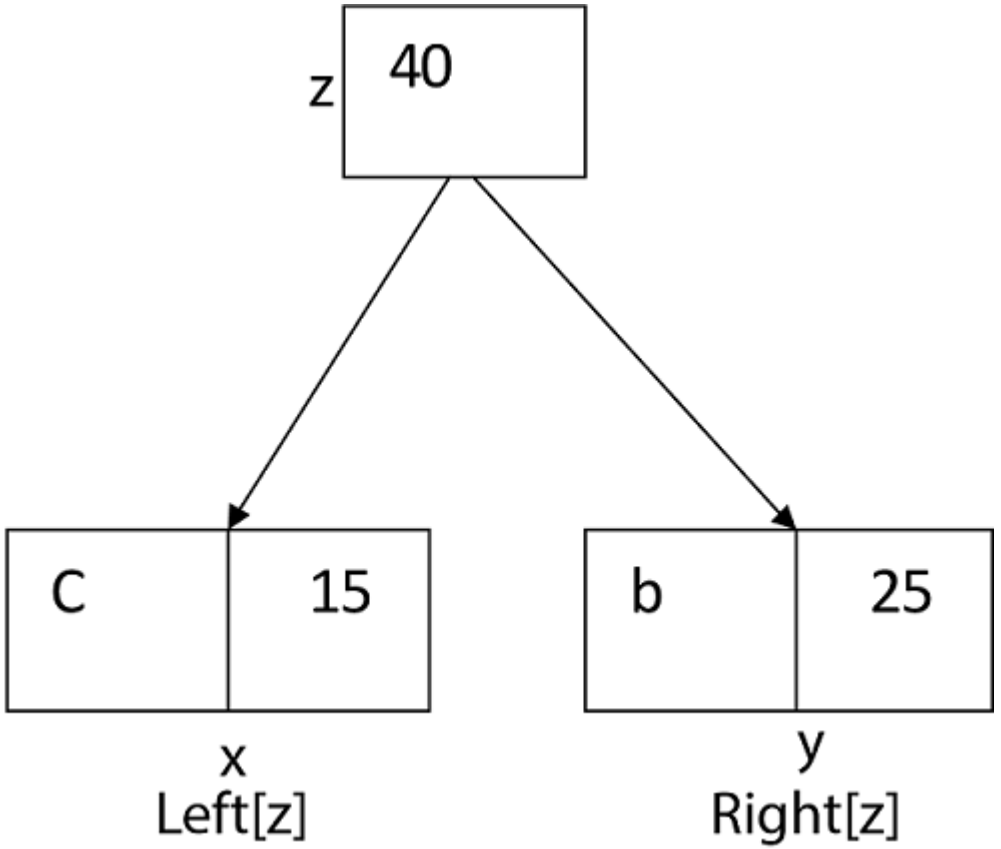
Left [z] ← x

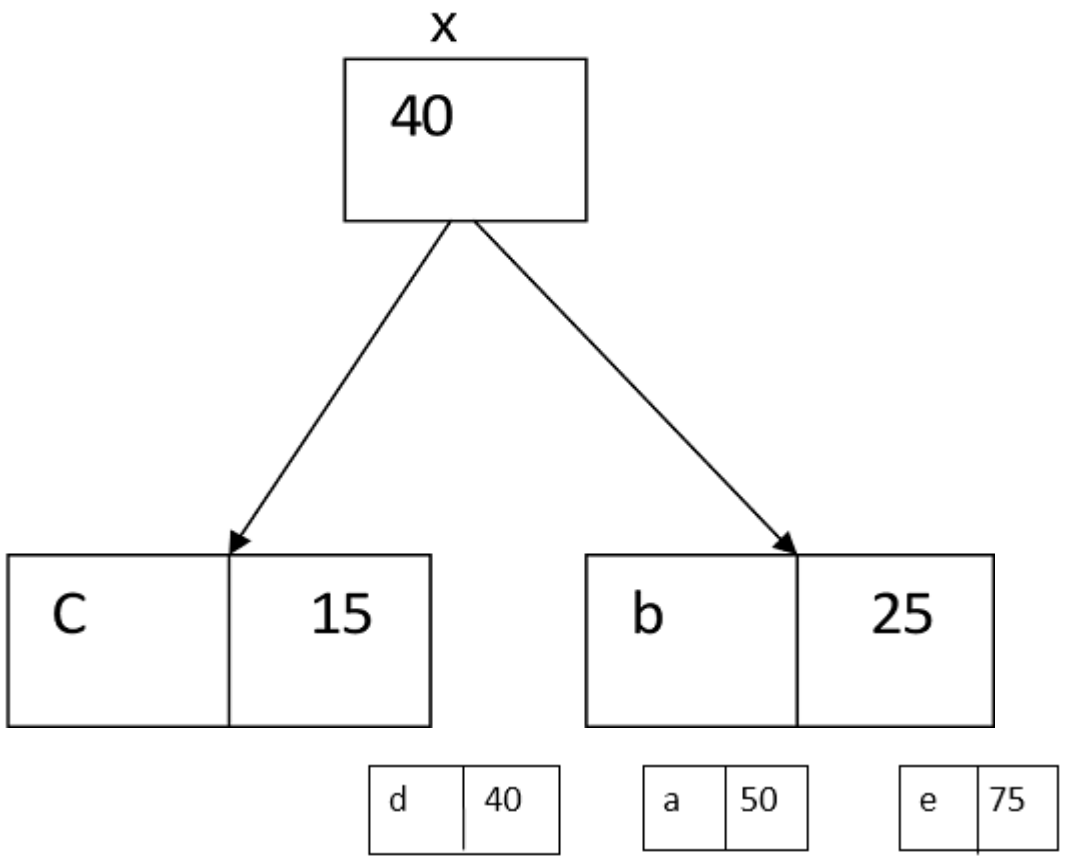
Right [z] ← y

f (z) ← f (x) + f (y) = 15 + 25

f (z) = 40







z ← Allocate node

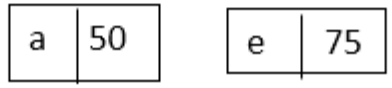
x ← 40

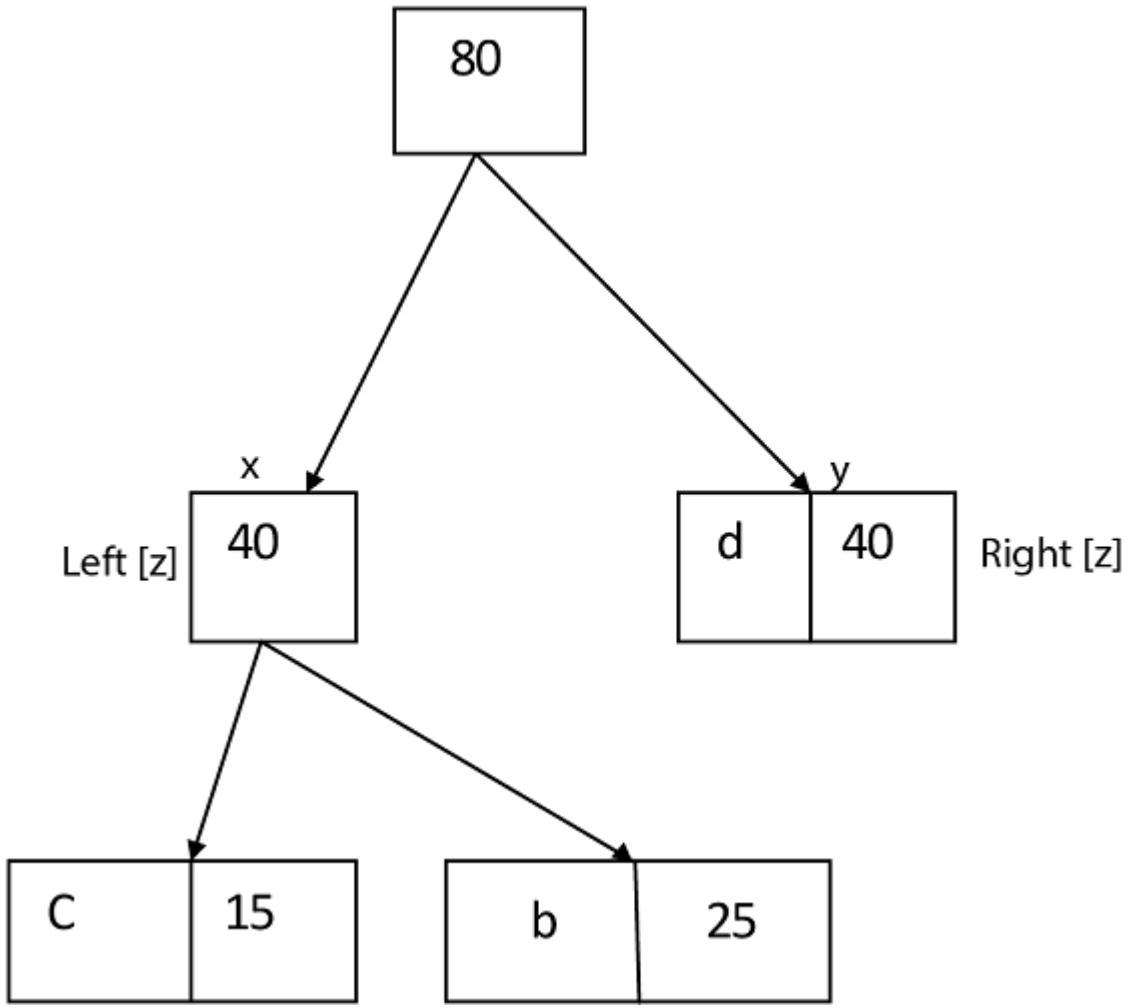
y ← 40

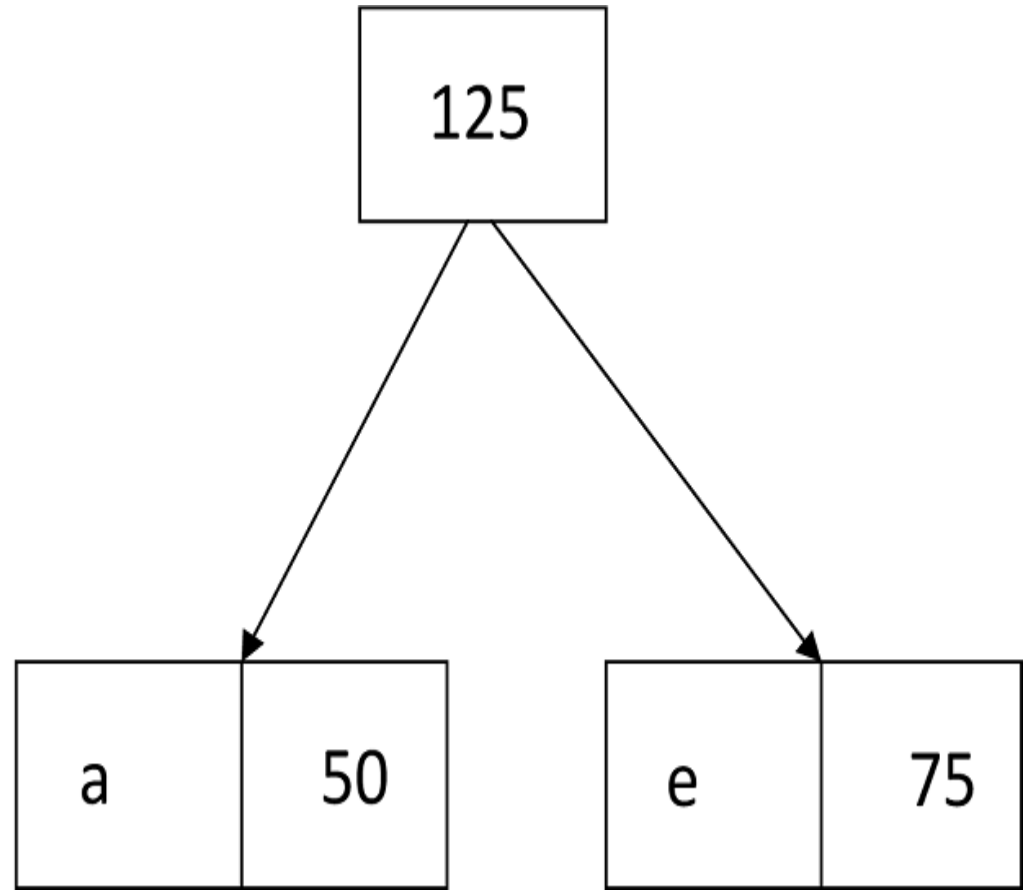
left [z] ← x

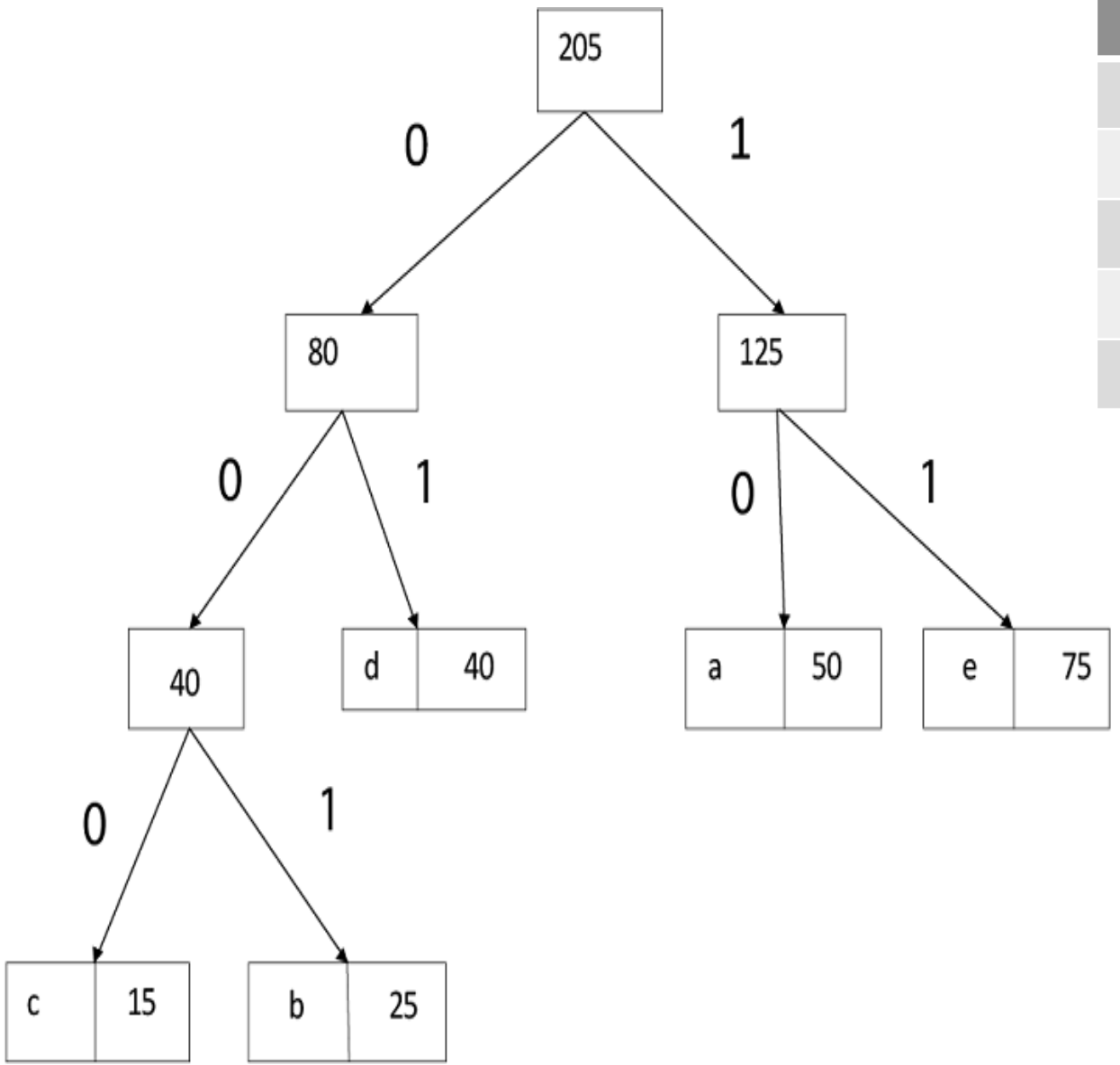
right [z] ← y

f (z) = 40 + 40 = 80




80
x
y
Left [z]
40
d
40
Right [z]
C
15
b
25


125
a
50
e
75



Character	Prefix (Variable prefix)
a	10
b	001
c	000
D	01
E	11