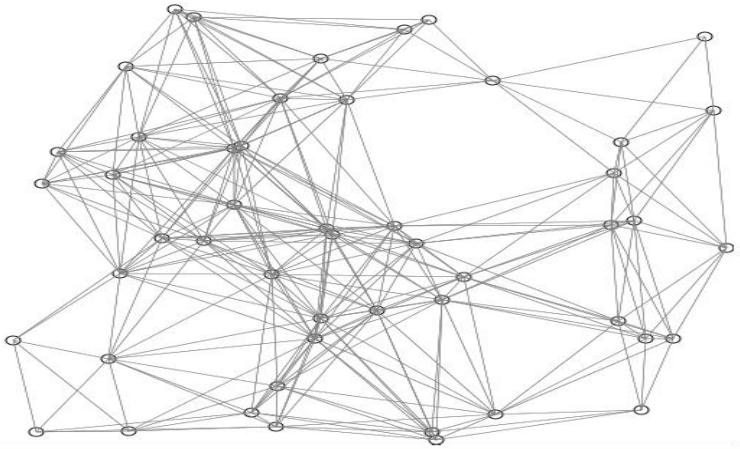
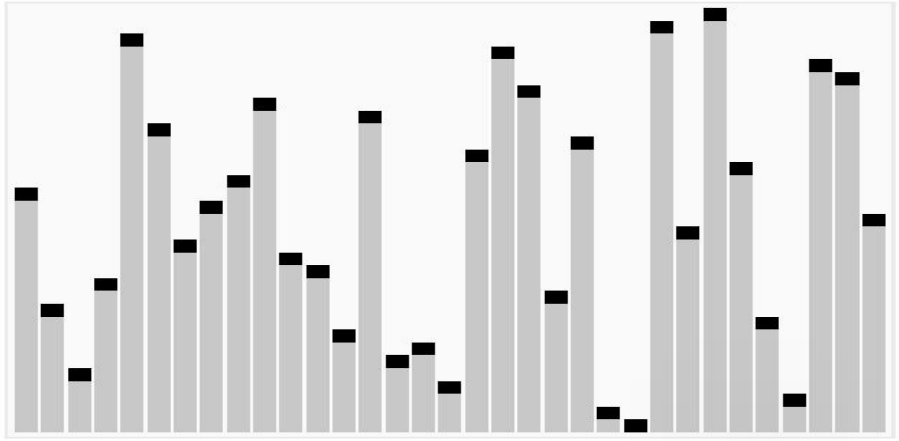
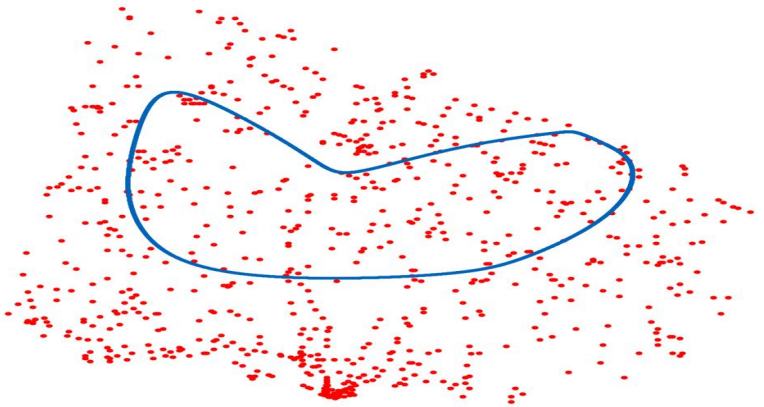
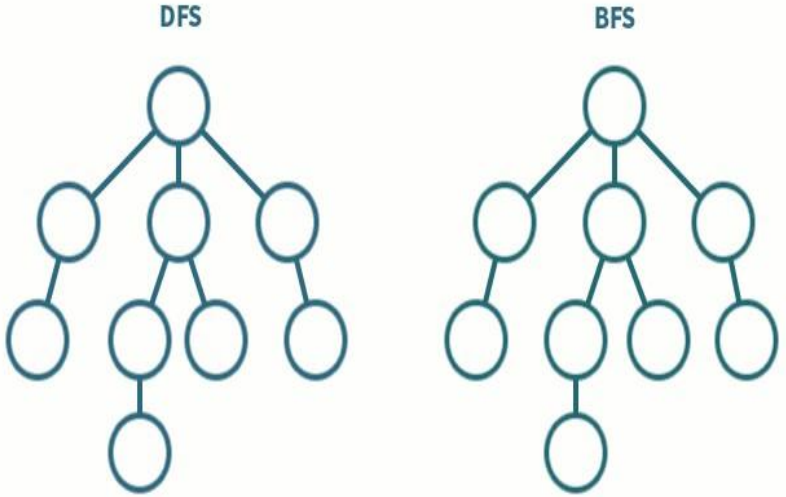








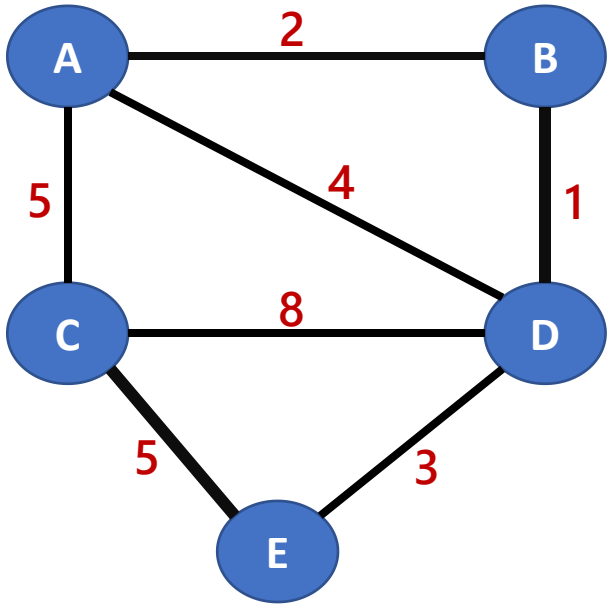
Design and Analysis of Algorithm

Dr. Supriyo Mandal,
Ph.D. (IIT Patna)
Postdoc (ZBW, University of Kiel, Germany)

- **Minimum Spanning Tree**
  - **Prim's Algorithm**
  - **Kruskal's Algorithm**
- **Single source shortest path**
  - **Dijkstra's Algorithm**

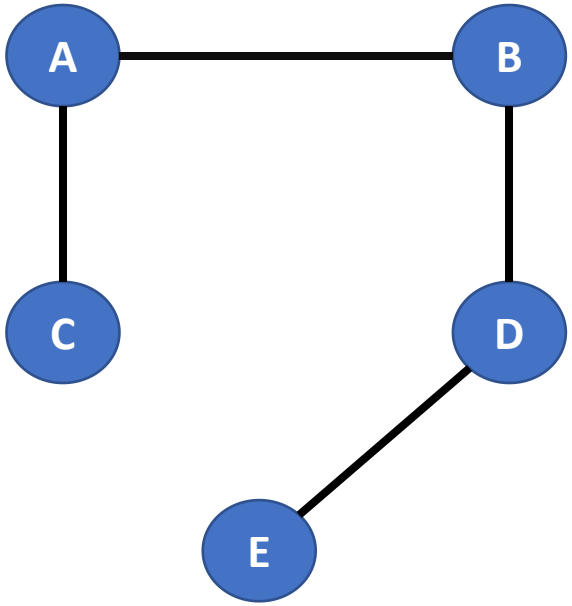
Basic Concepts

Weighted Graph: *A weight(w_i) is associated with every edge of graph*

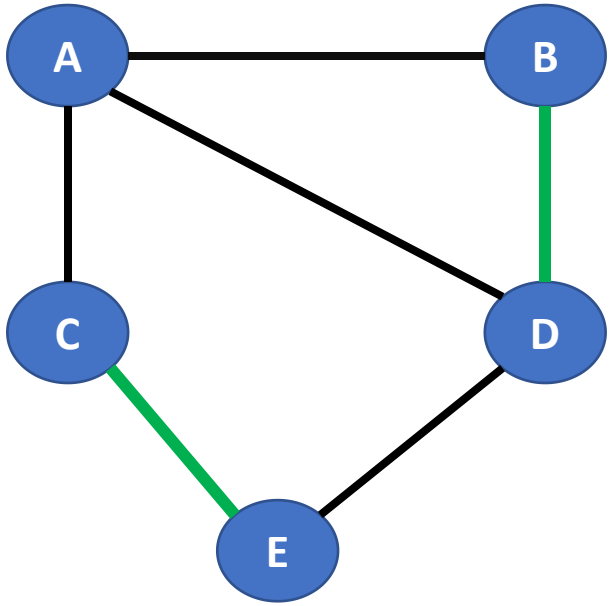


Basic Concepts

Acyclic Graph: A graph without cycle



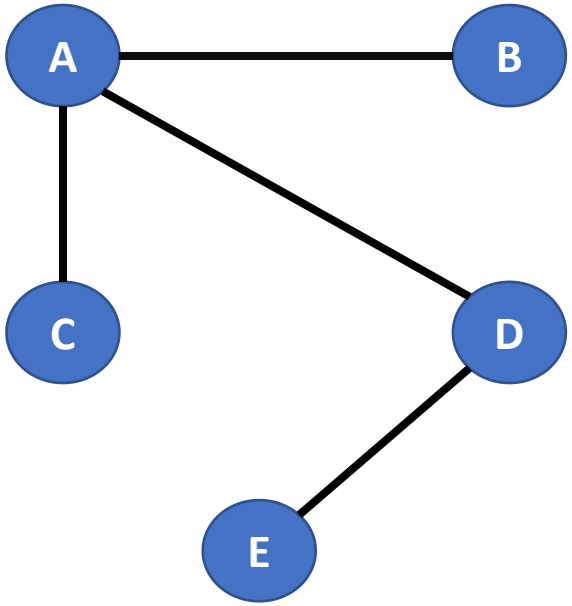
Acyclic Graph



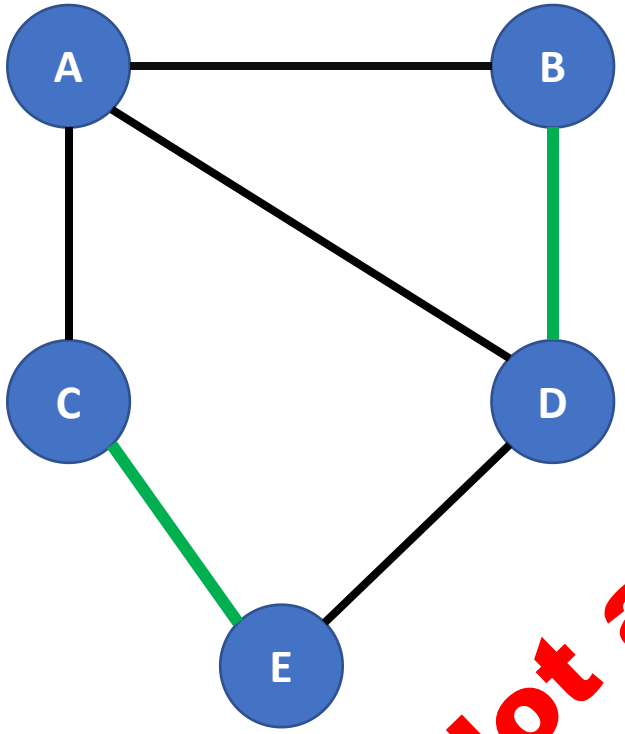
Cyclic Graph

Basic Concepts

Tree: A connected acyclic graph



Tree



Not a Tree

Basic Concepts

Spanning Tree

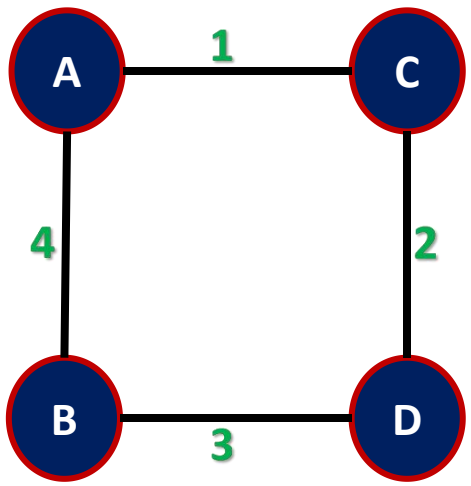
A spanning tree of a graph G=<V,E> is a subgraph G' of G such that

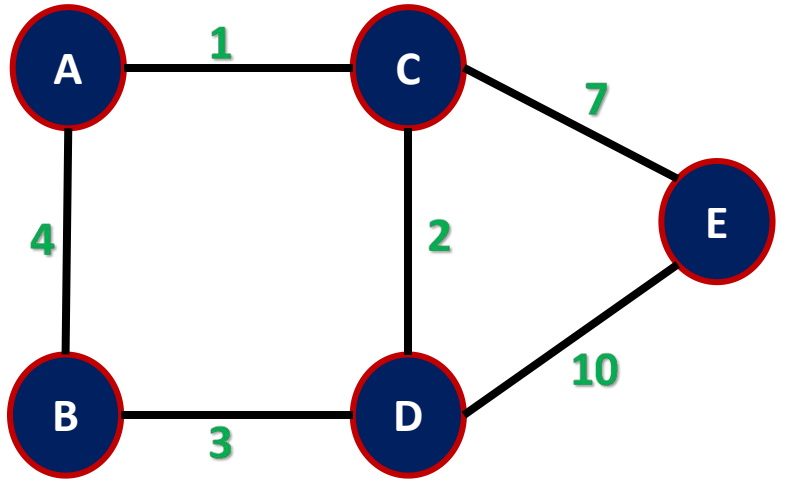
1. G' is a Tree and
2. G' spans(covers) all the vertices of G

Basic Concepts

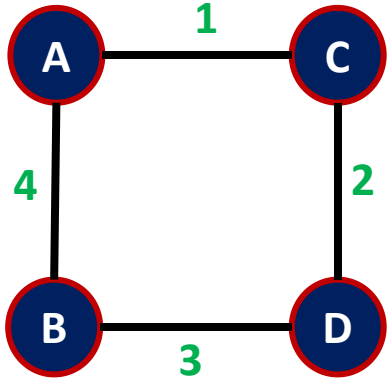
Weighted Connected Undirected Graph:

A connected undirected graph in which every edge has +ve weight

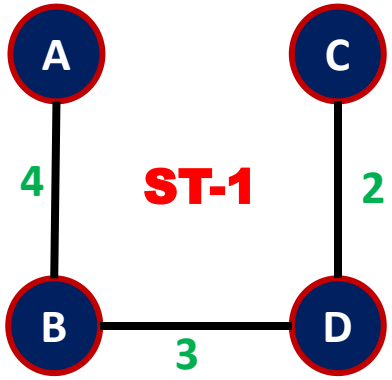




Minimum Spanning Tree



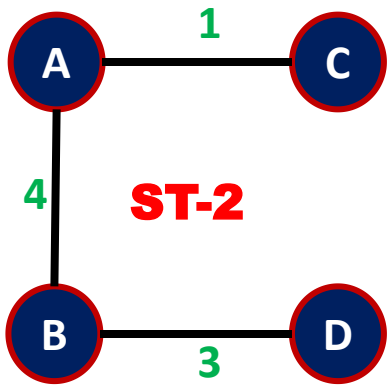
MST



ST-1

Cost

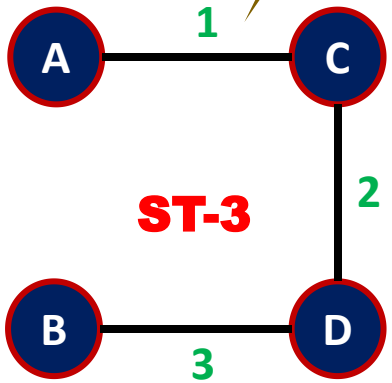
2 + 3 + 4 = 9



ST-2

Cost

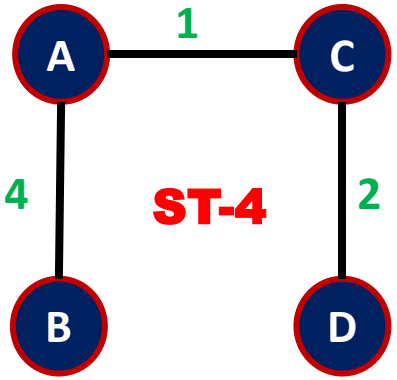
1 + 3 + 4 = 8



ST-3

Cost

2 + 3 + 1 = 6



ST-4

Cost

2 + 1 + 4 = 7

Application of MST

- Constructing highways or railroads spanning several cities

You want to supply a set of houses with

- Electric Power
- Water
- Telephone lines
- Sewage lines

How to find MST of a Graph

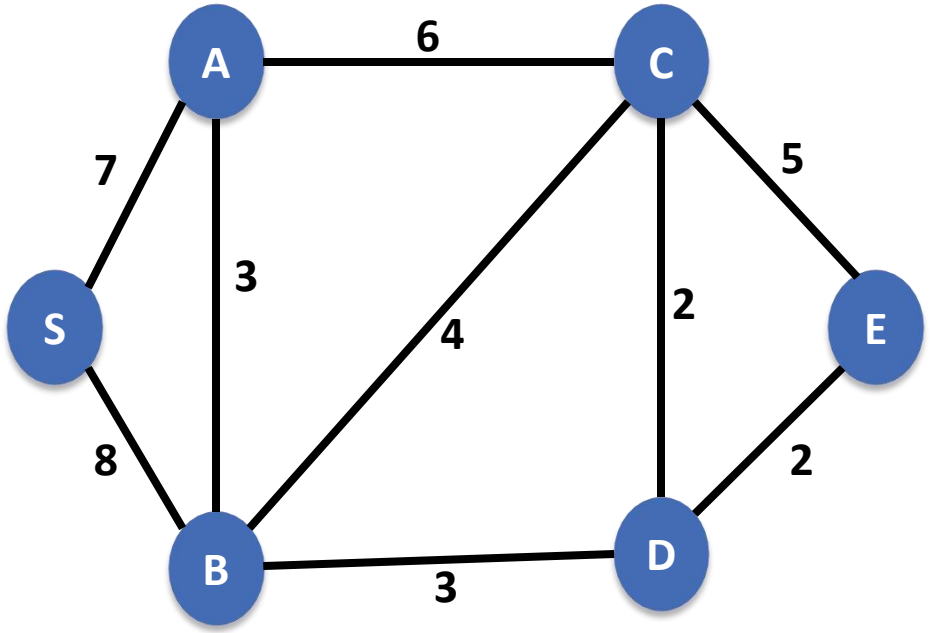
MST of a graph can be computed using

- Kruskal's Algorithm(1956)
- Prim's Algorithm(1959)

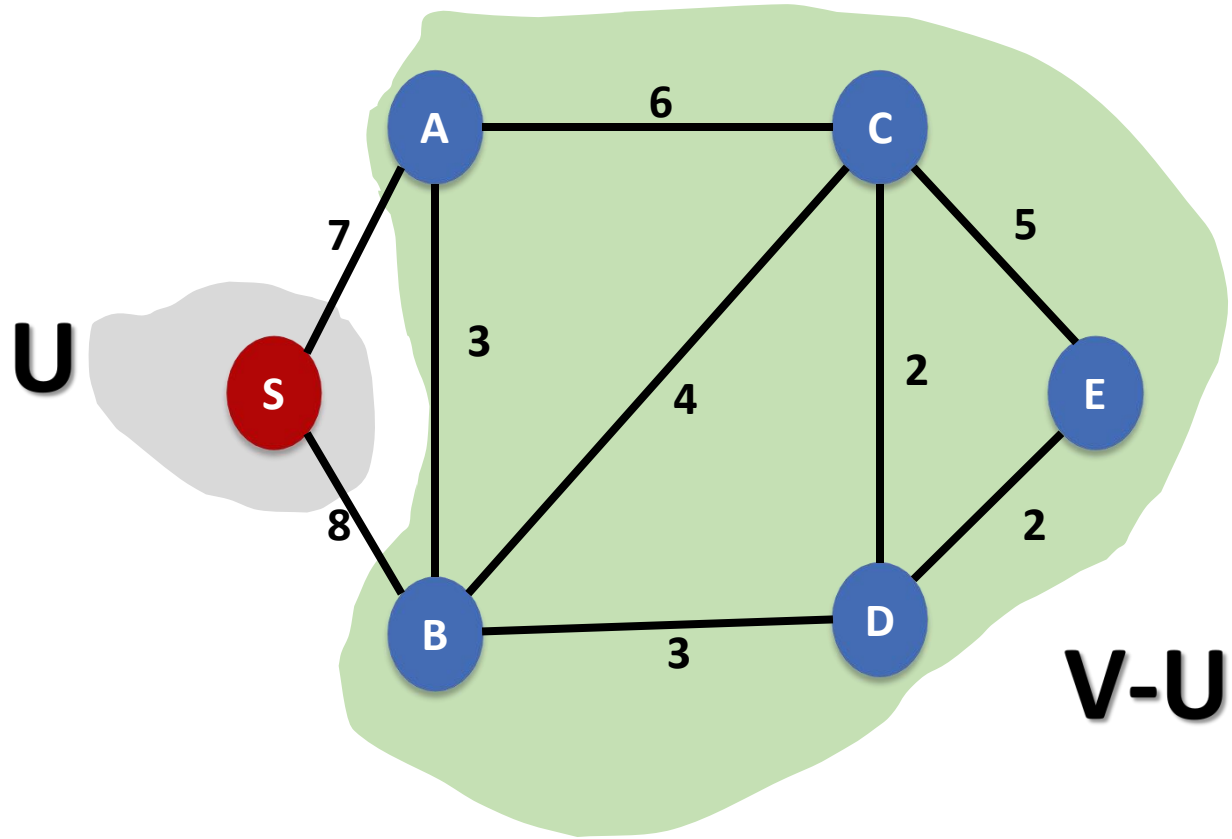
Prim's Algorithm

Consider the following graph

Assume S is source vertex

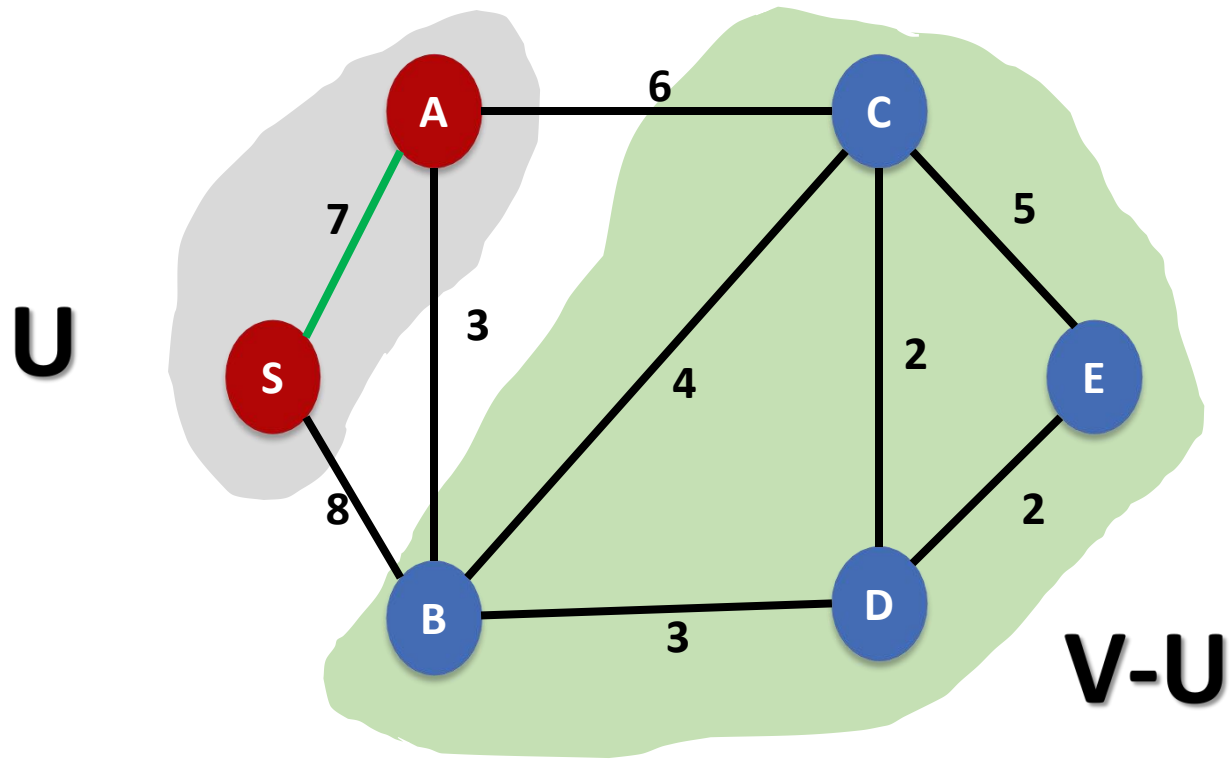


Prim's Algorithm



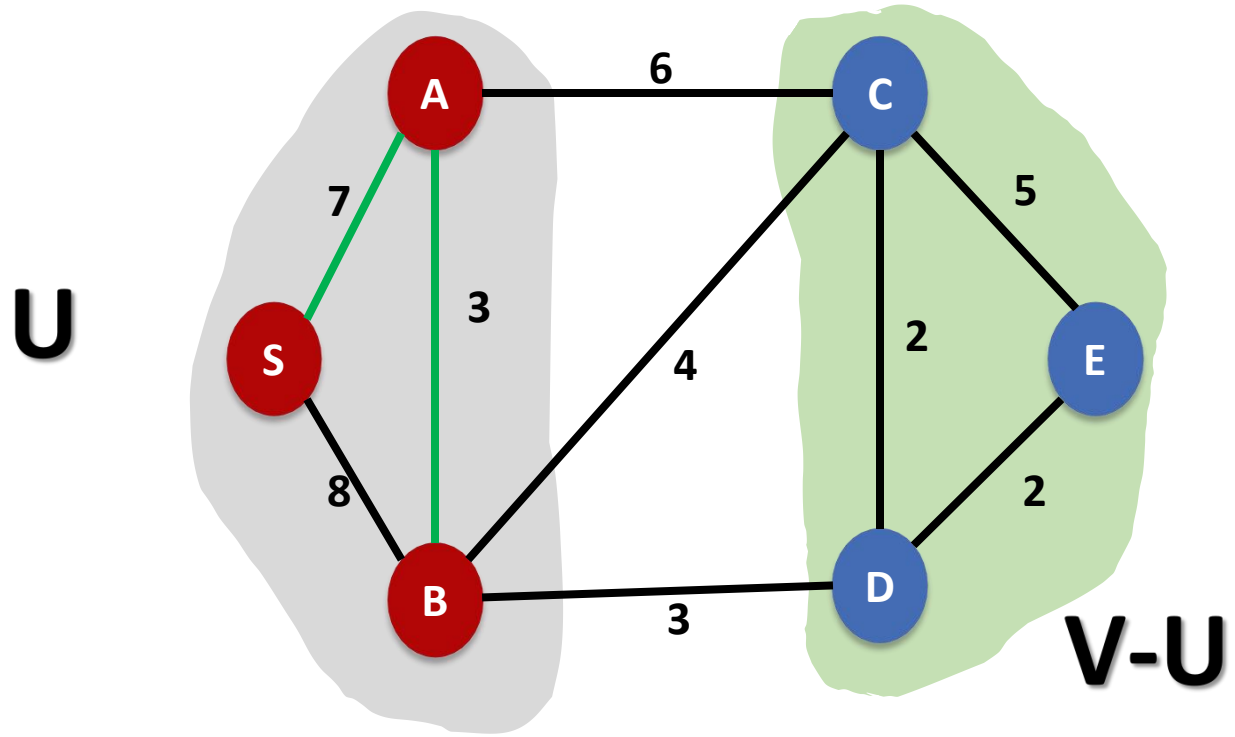
Pick the **Lowest** cost edge connecting U and V-U

Prim's Algorithm



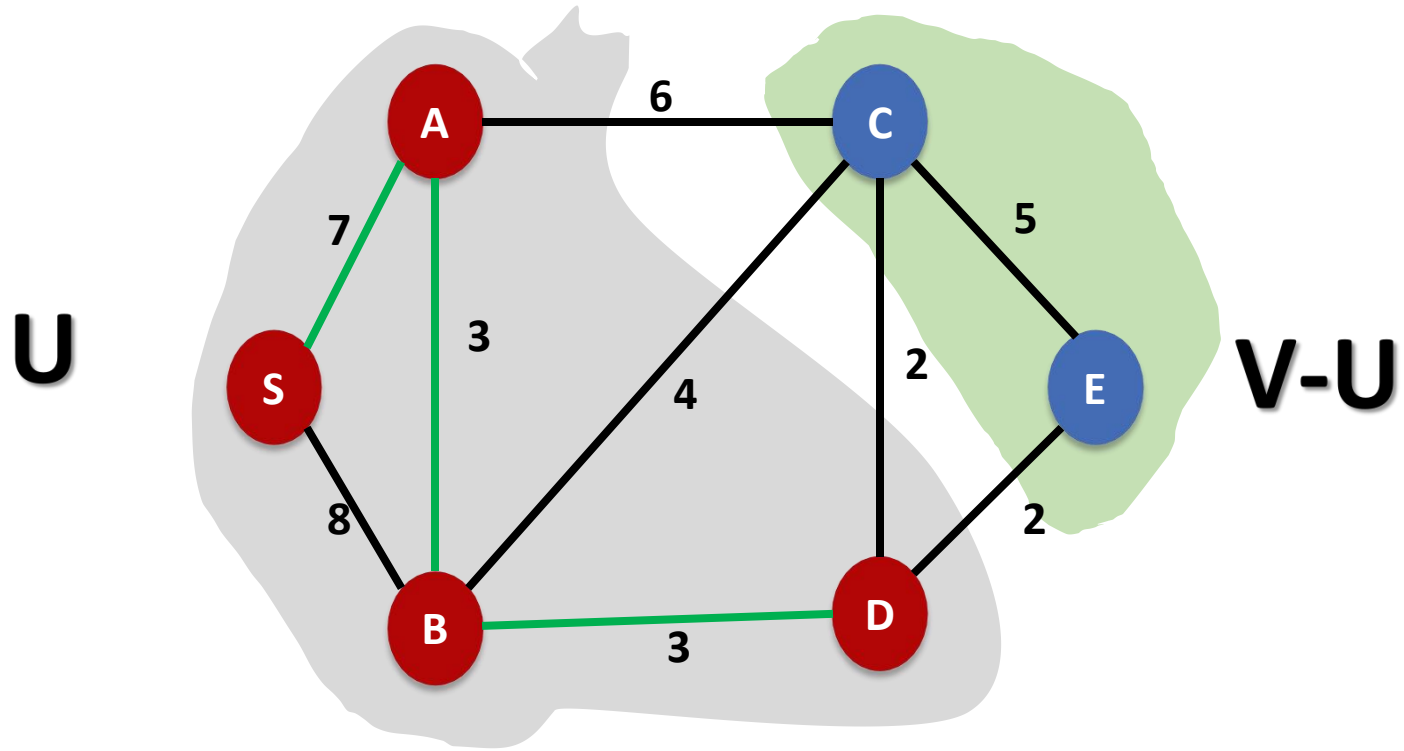
Pick the **Lowest** cost edge connecting U and V-U

Prim's Algorithm



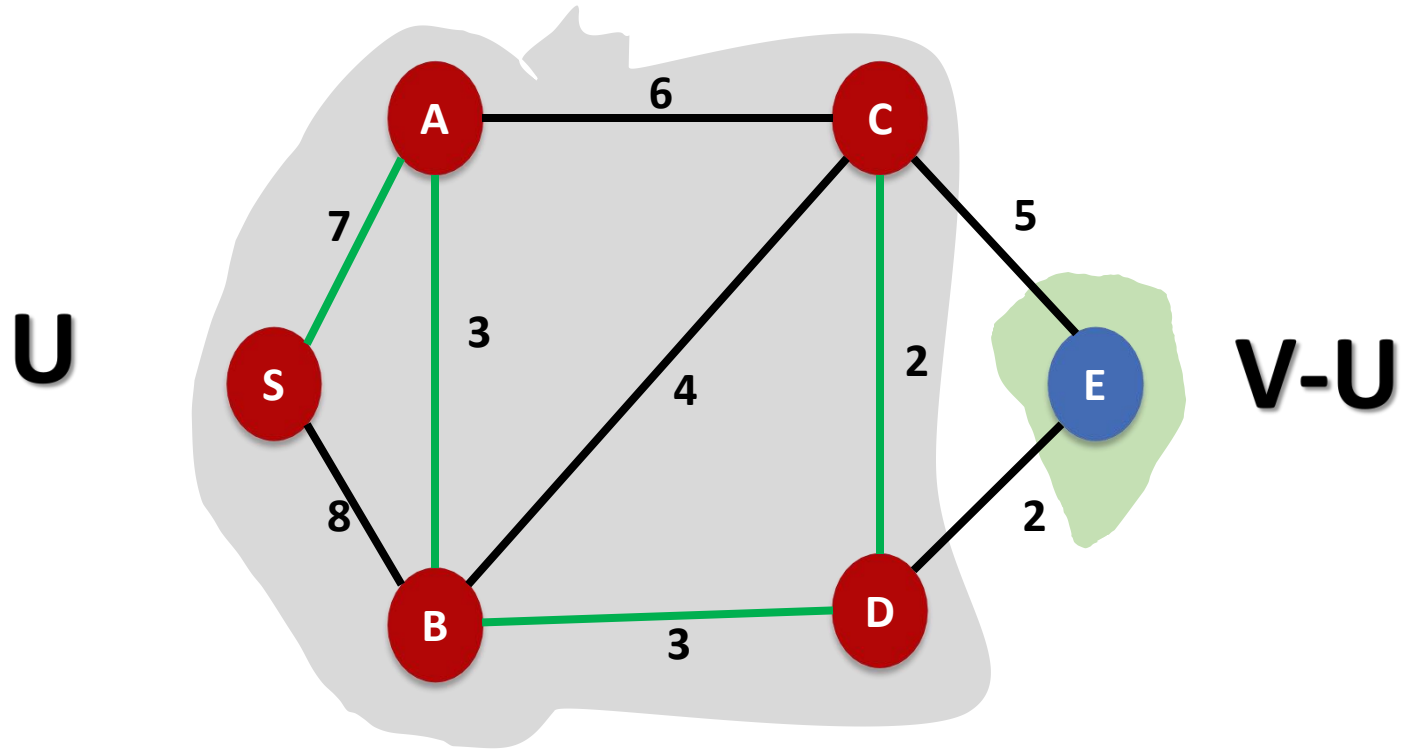
Pick the Lowest cost edge connecting U and V-U

Prim's Algorithm



Pick the Lowest cost edge connecting U and V-U

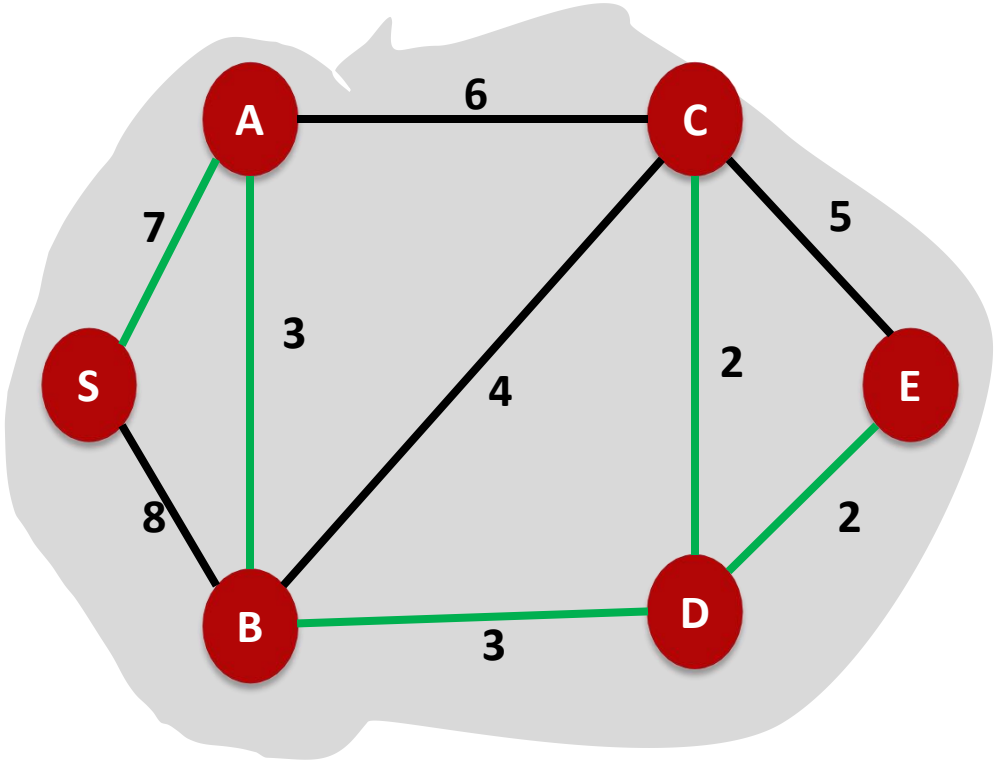
Prim's Algorithm



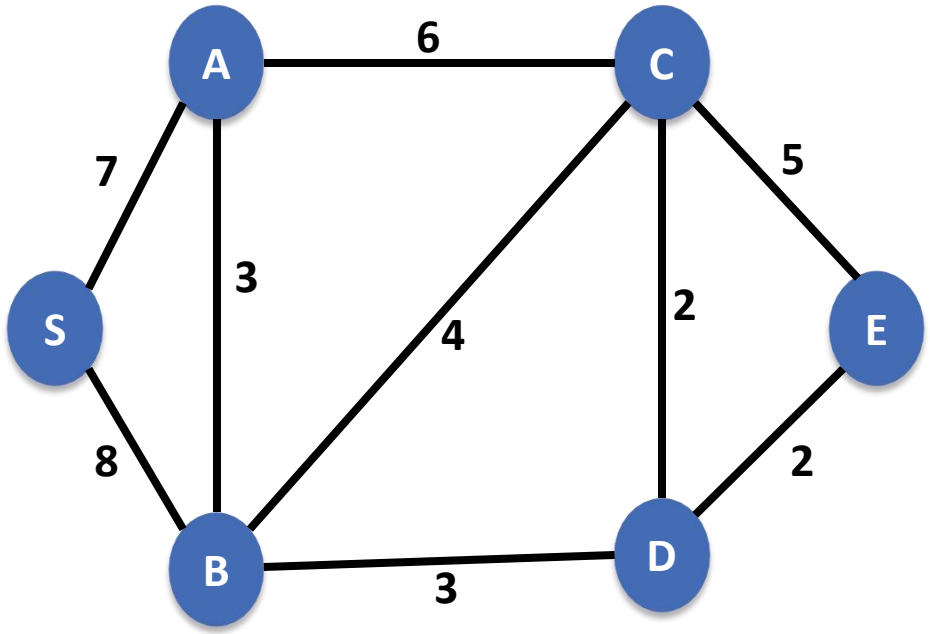
Pick the Lowest cost edge connecting U and V-U

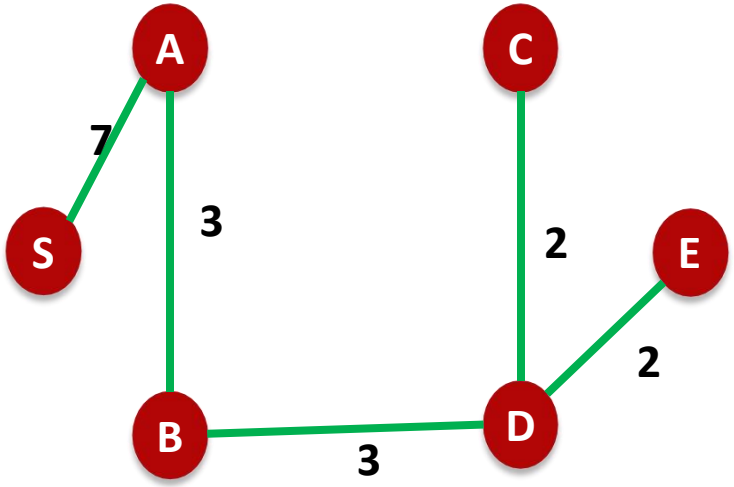
Prim's Algorithm

U



Prim's Algorithm

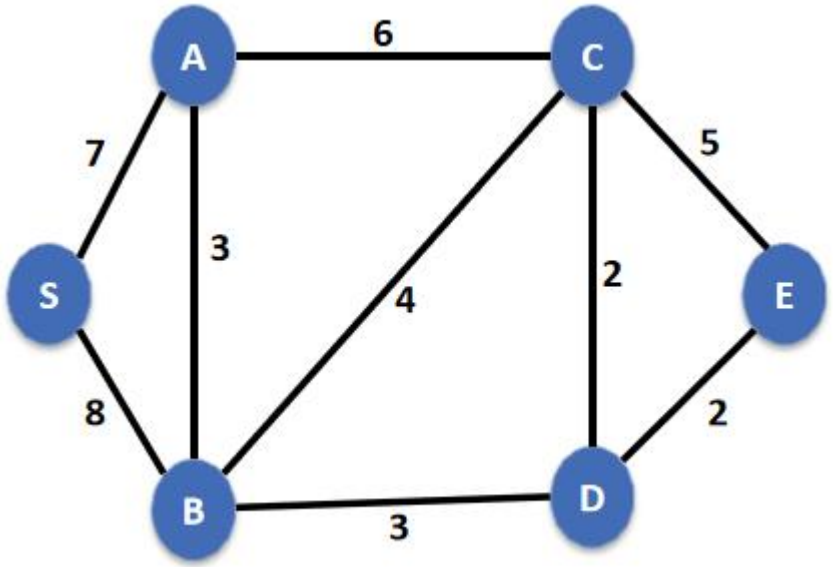


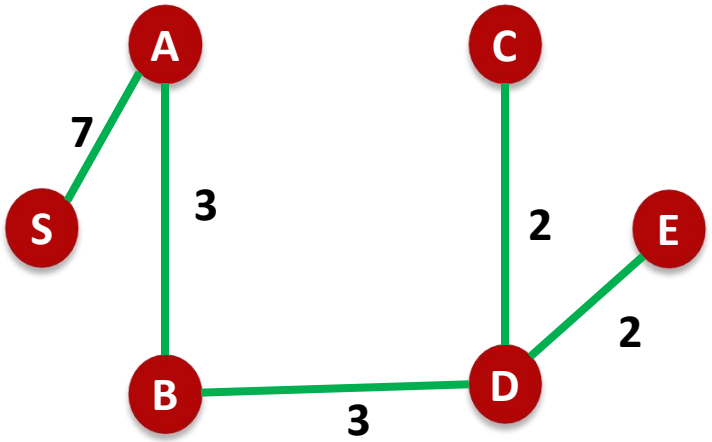


MST

The Cost = 7 + 3 + 3 + 2 + 2 = **17**

Prim's Algorithm





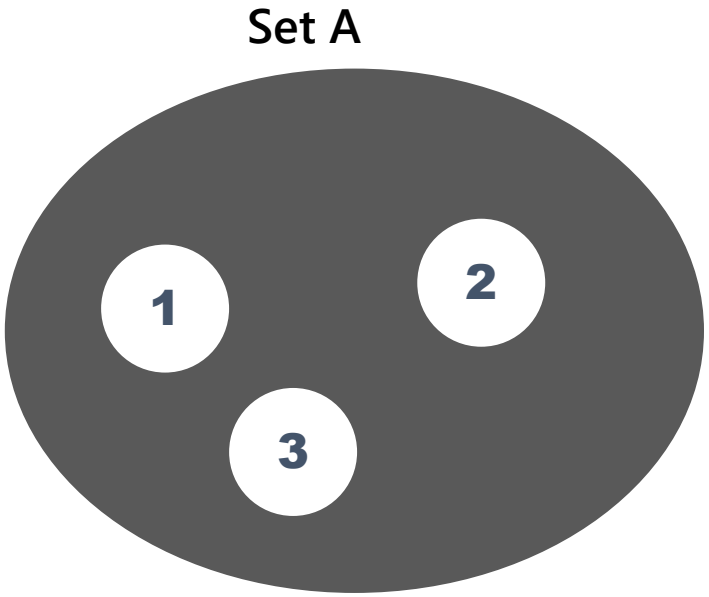

```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7      INSERT(Q, u)
 8  while Q ≠ ∅
 9      u = EXTRACT-MIN(Q)          // add u to the tree
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))
```

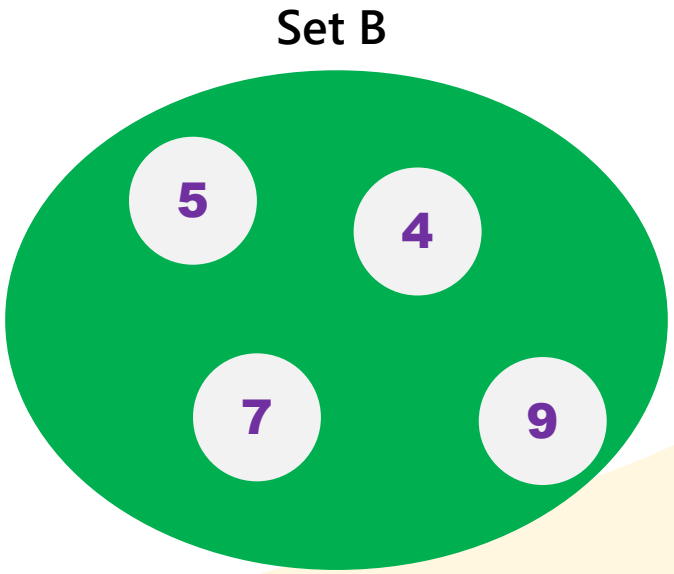
O(V) + O(V) + O(VlogV) + O(Elogv) = O(E+V)logV =O(Elogv)

Disjoint Set Data Structure

Disjoint Set Data Structure

Disjoint Set





A and B are disjoint if
$A \cap B = \emptyset$

Set A and Set B are partition of set {1, 2, 3, 4, 5, 7, 9}

Disjoint Set Data Structure

Invented By Michel Fischer in 1964

- It is a data structure that stores a collection of disjoint (non-overlapping) sets
- it stores a partition of a set into disjoint subsets.
- It support following operations
  - ✓ Makeset
  - ✓ union/merge
  - ✓ Find

Also known as Union Find OR Merge Find Data Structure

Disjoint Set Data Structure

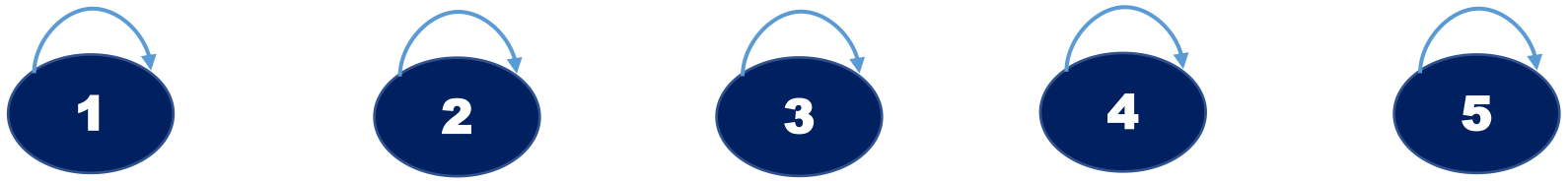
MakeSet(x) : Create a singleton set {x}

```
void MakeSet(x)
    {
        parent[x]= x
    }
```

Cost = O(1)

Disjoint Set Data Structure

makeset(1) makeset(2) makeset(3) makeset(4) makeset(5)



Element	1	2	3	4	5
Parent	1	2	3	4	5

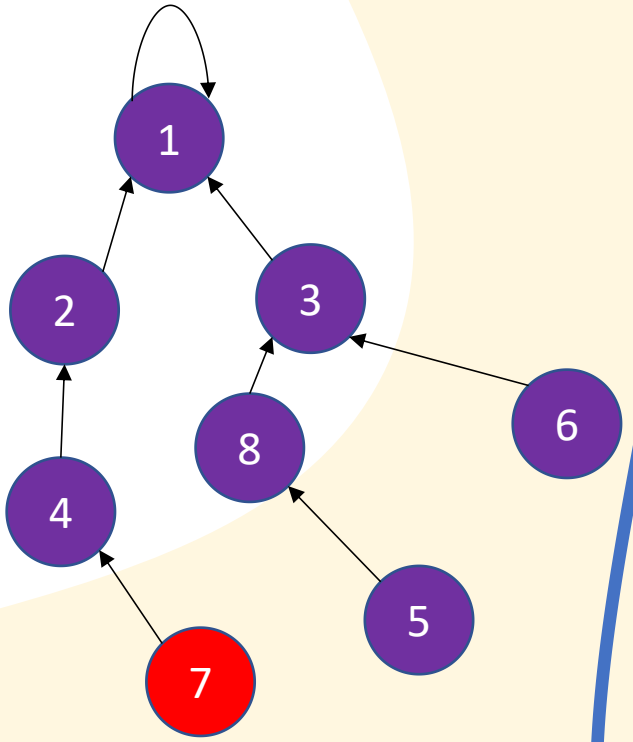
Disjoint Set Data Structure

findset(x) : returns the parent (**representative/leader**) of x

```
int findset(int x)
   {
       if (x == parent[x])
           return x;
       return findset(parent[x]);
   }
```

Disjoint Set Data Structure

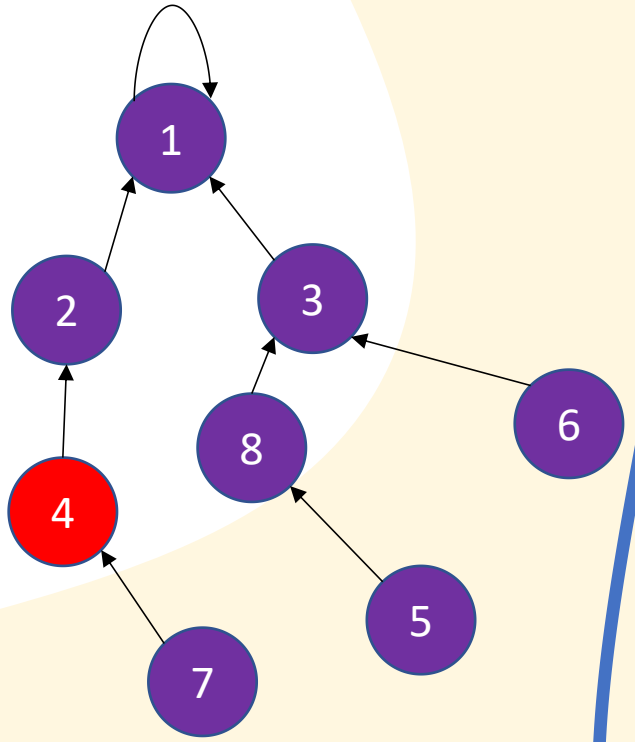
```
int findset(int x)
    {
        if (x == parent[x])
            return x;
        return findset(parent[x]);
    }
```



Findset(7) = ?

Disjoint Set Data Structure

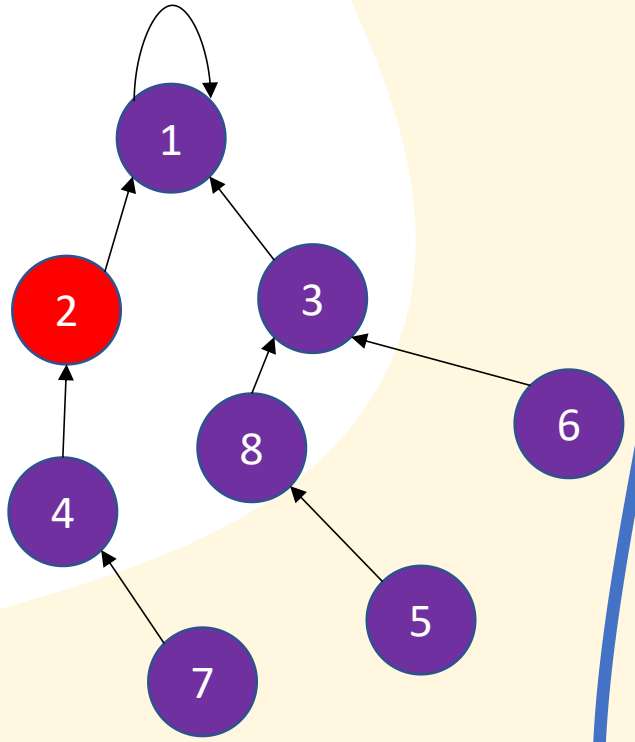
```
int findset(int x)
    {
        if (x == parent[x])
            return x;
        return findset(parent[x]);
    }
```



Findset(7) → findset(4)→

Disjoint Set Data Structure

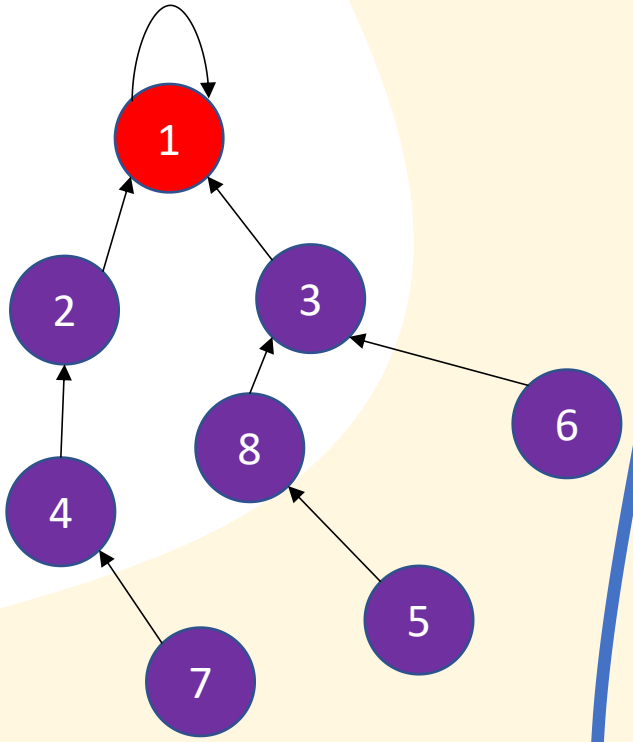
```
int findset(int x)
    {
        if (x == parent[x])
            return x;
        return findset(parent[x]);
    }
```



Findset(7) → findset(4)→findset(2)

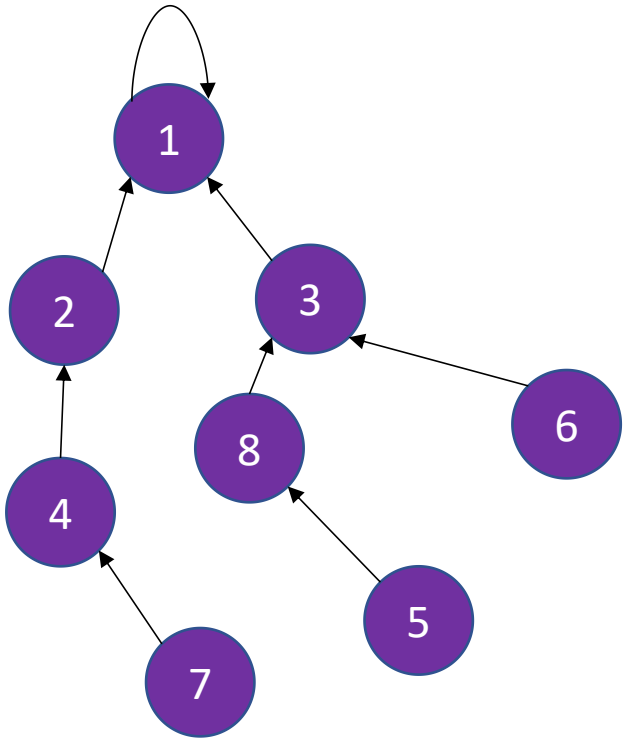
Disjoint Set Data Structure

```
int findset(int x)
    {
        if (x == parent[x])
            return x;
        return findset(parent[x]);
    }
```



findset(7) → findset(4)→findset(2)→findset(1) =1

Disjoint Set Data Structure



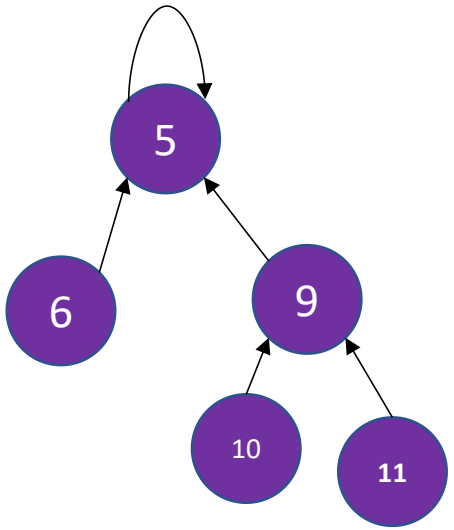
Findset(7) = 1

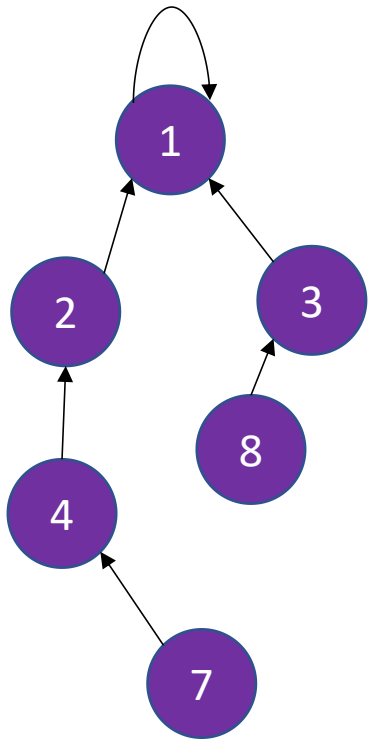
Findset(5) = 1

7 and 5 are member of SAME set

Observation1: If find(a) = find(b) then a and b belongs from same set

Disjoint Set Data Structure





Findset(7) = 1

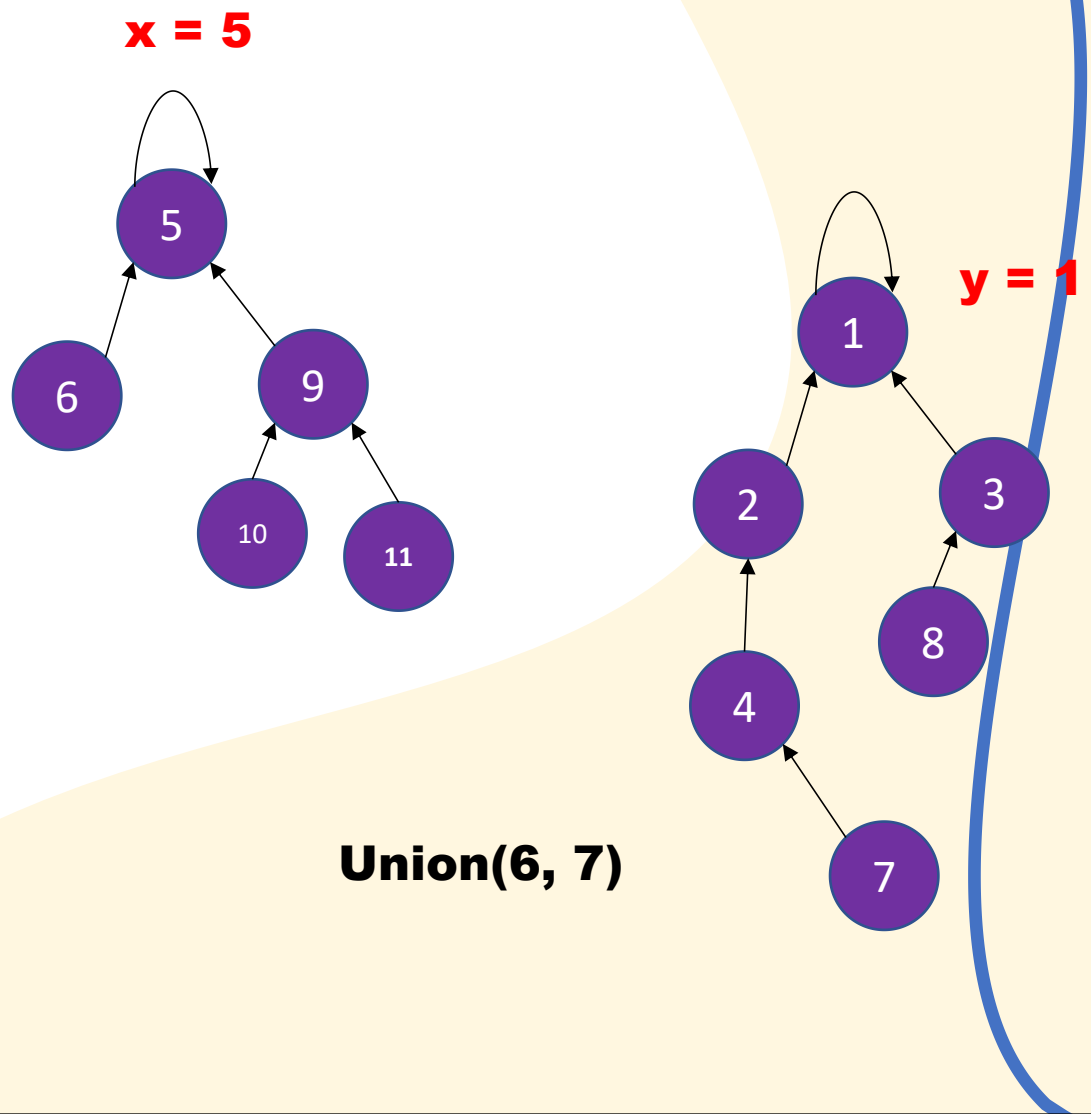
Findset(10) = 5

7 and 10 are member of DIFFERENT set

Observation2: If find(a) ≠ find(b) then a and b belongs from Different set

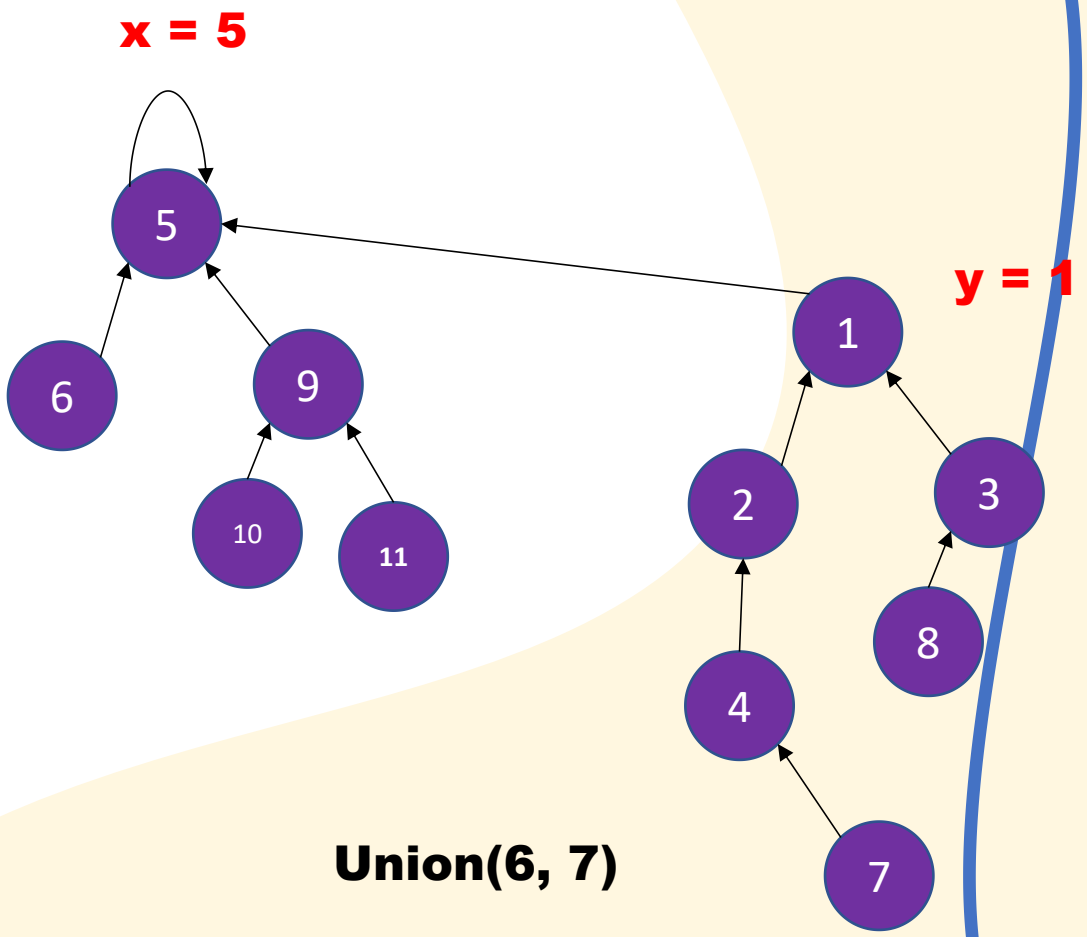
Disjoint Set Data Structure

```
void union(int a, int b)
        {
                x = findset(a);
                y = findset(b);
                if (x != y)
                    parent[y] = x;
        }
```



Disjoint Set Data Structure

```
void union(int a, int b)
        {
                x = findset(a);
                y = findset(b);
                if (x != y)
                    parent[y] = x;
        }
```



Kruskal's Algorithm

STEPS:

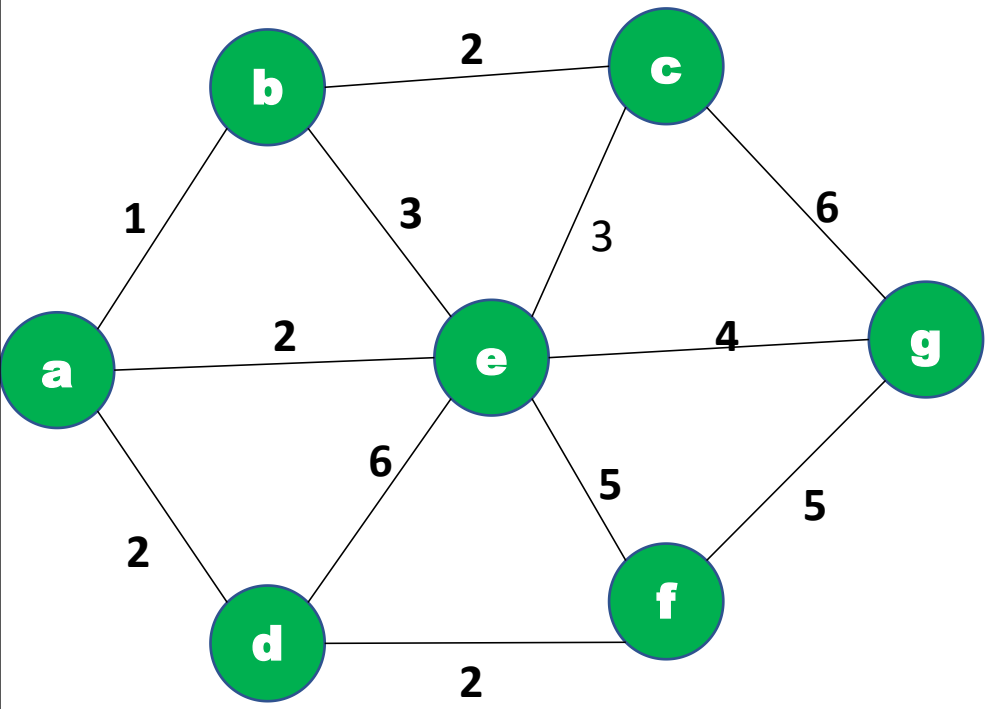
1. Initially, trees of the forest are the vertices (no edges).

2. In each step add the cheapest edge that does not create a cycle.

3. Continue until the forest 'merge to' a single tree

Kruskal's Algorithm

Finding MST using Kruskal's Algorithm



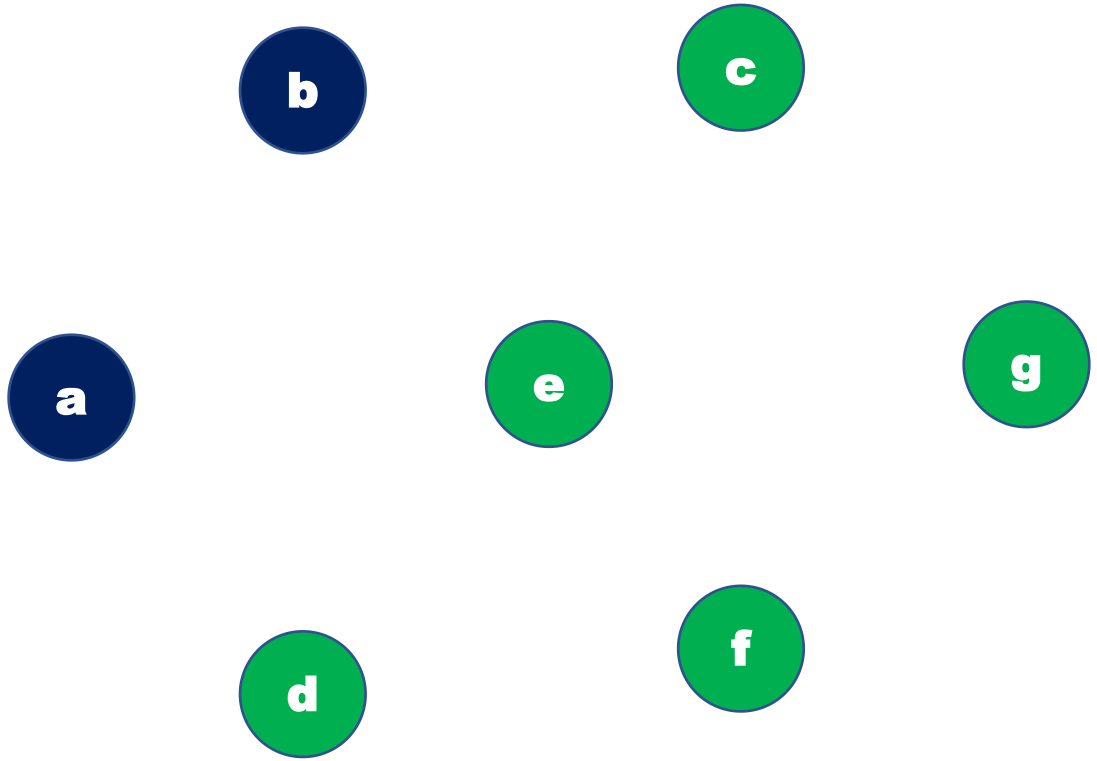
Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

SORTED

A = { }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

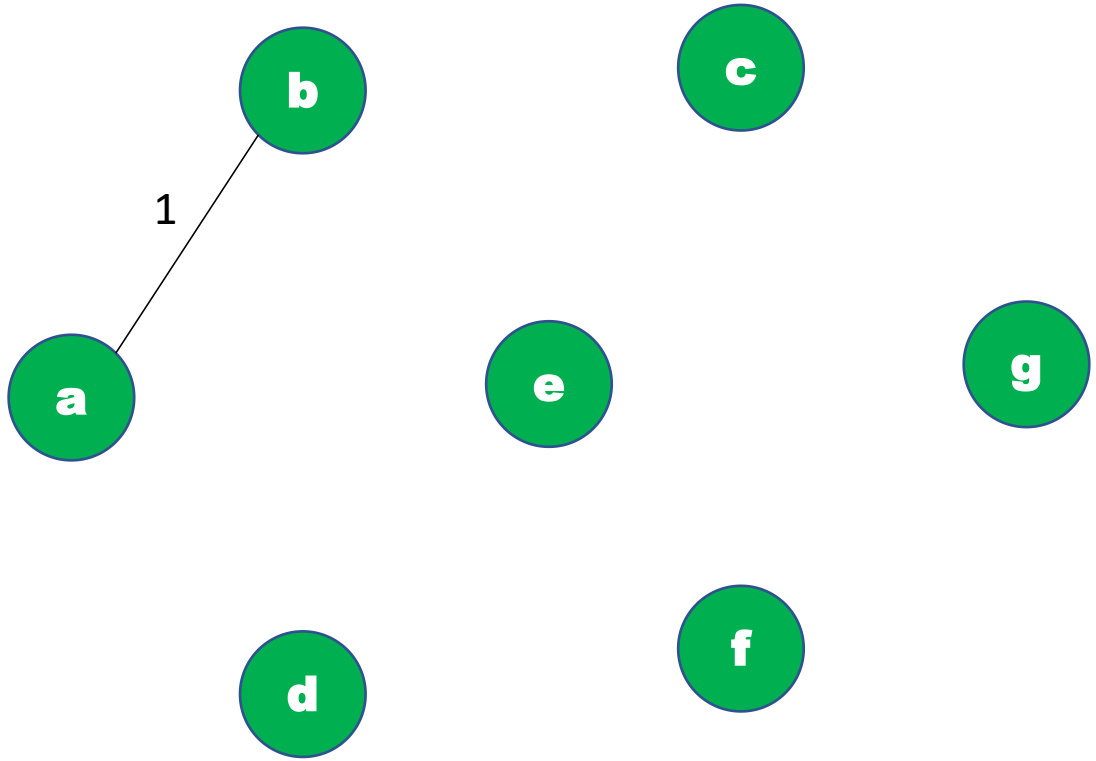


Number of connected components = 7

A = { }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

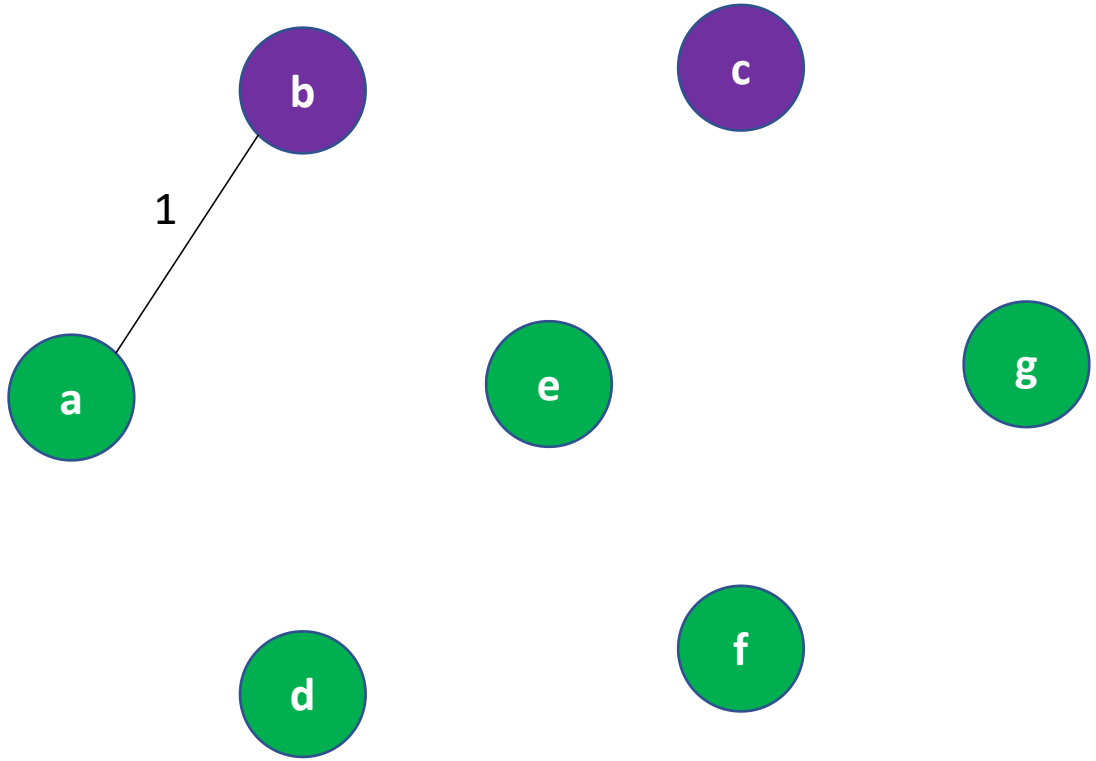


Number of connected components = 6

A = {ab }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

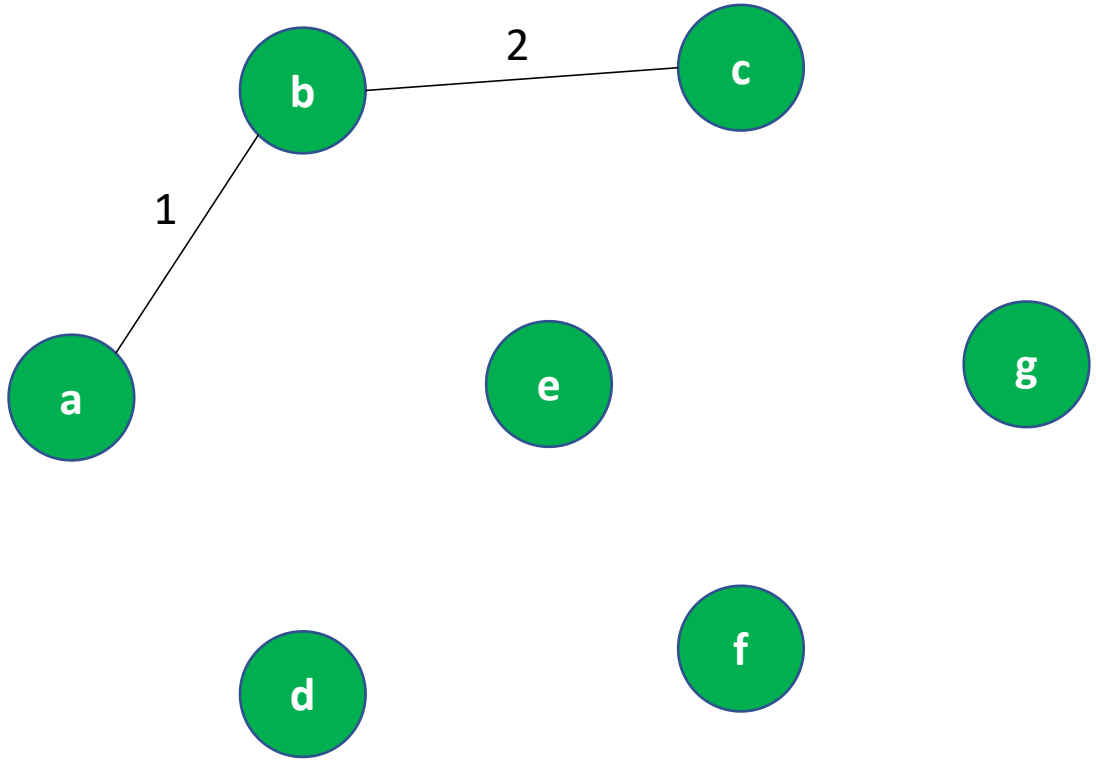


Number of connected components = 6

A = {ab }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

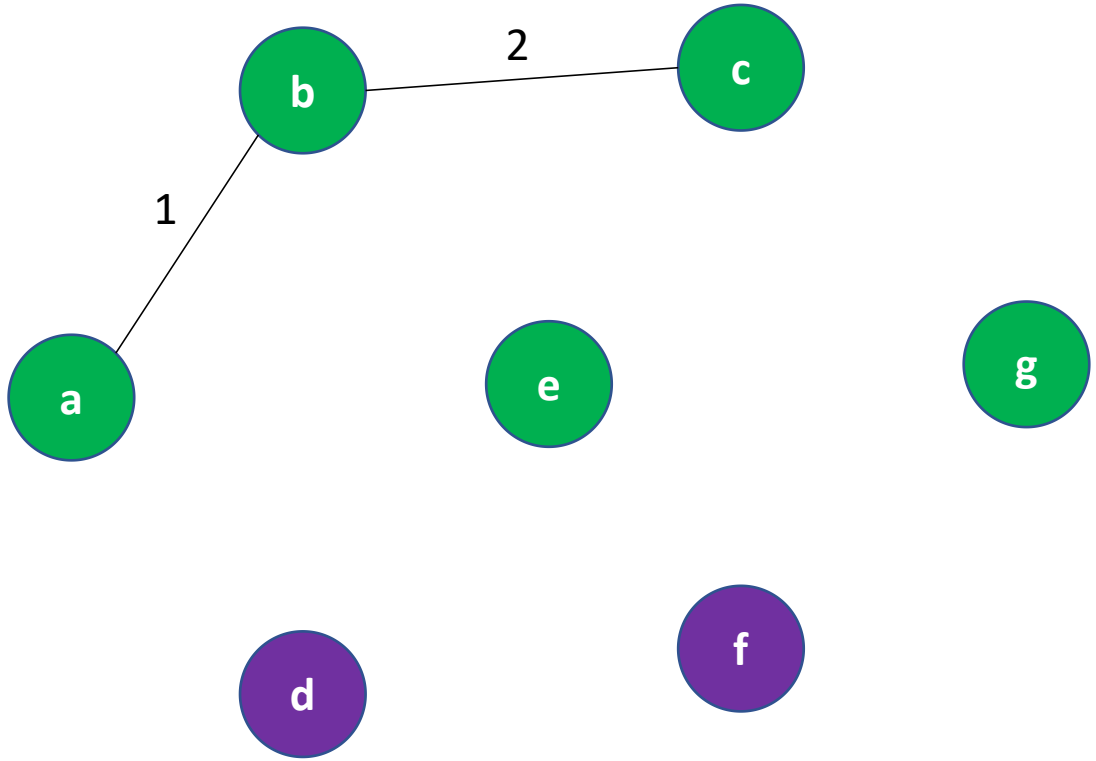


Number of connected components = 5

A = {ab, bc }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

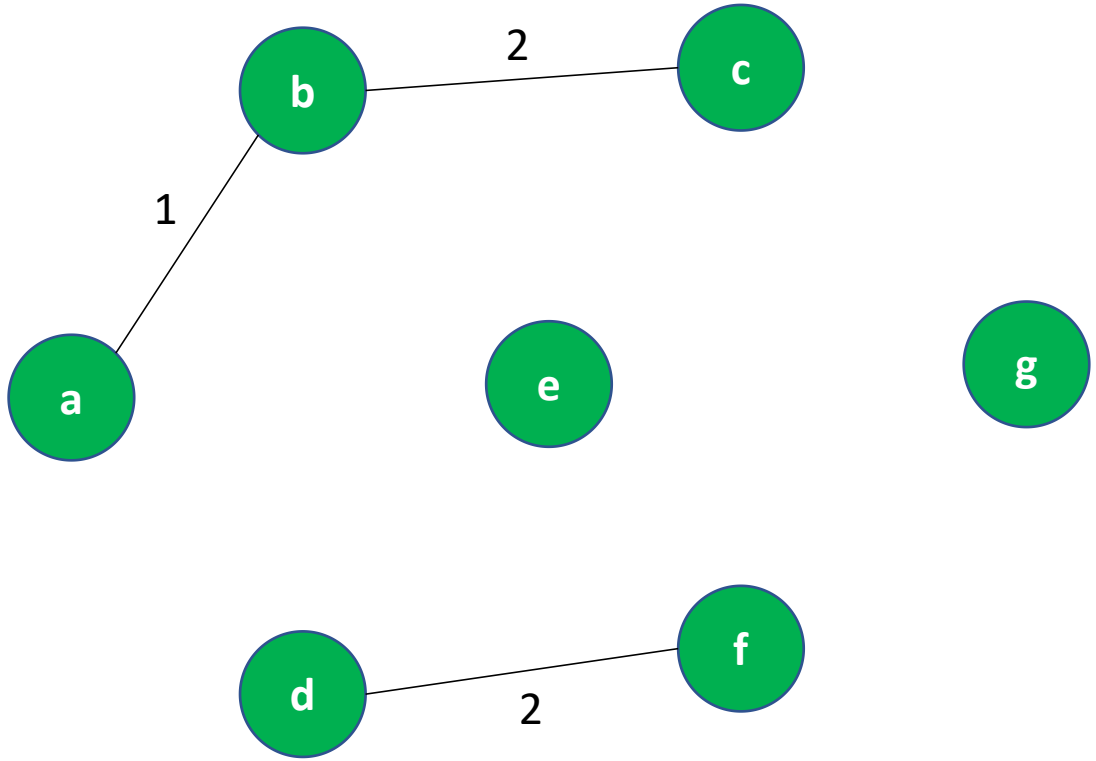


Number of connected components = 5

A = {ab, bc }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

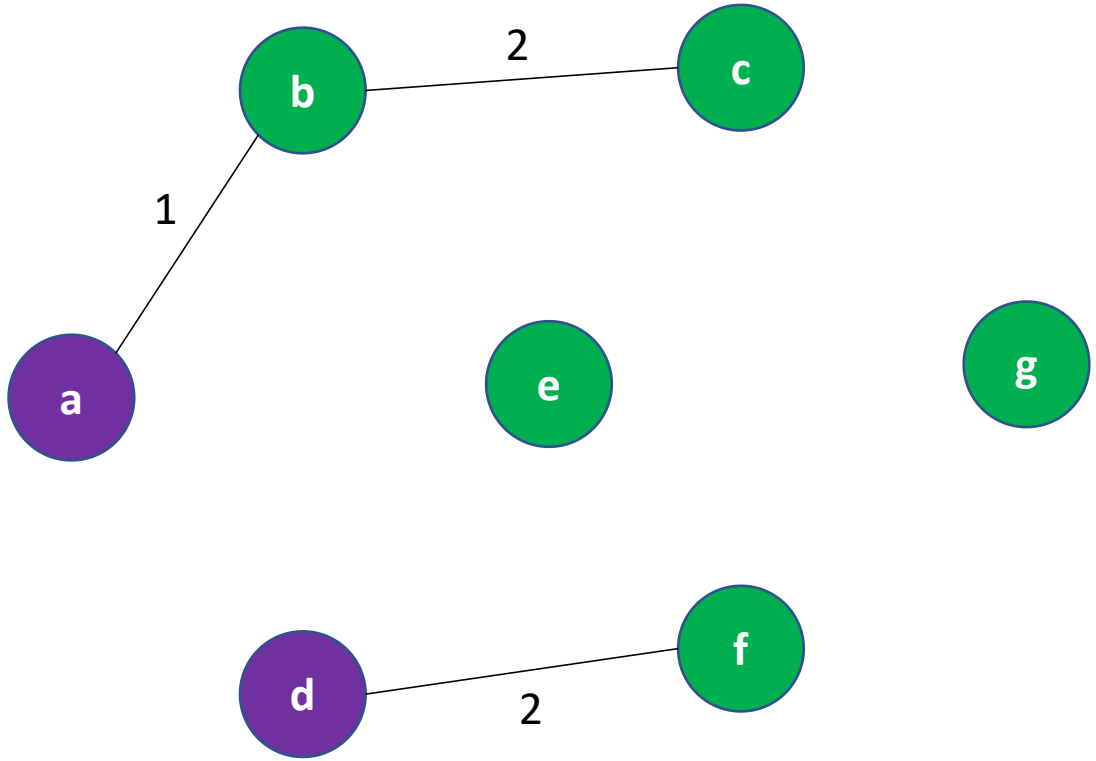


Number of connected components = 4

A = {ab, bc, df }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

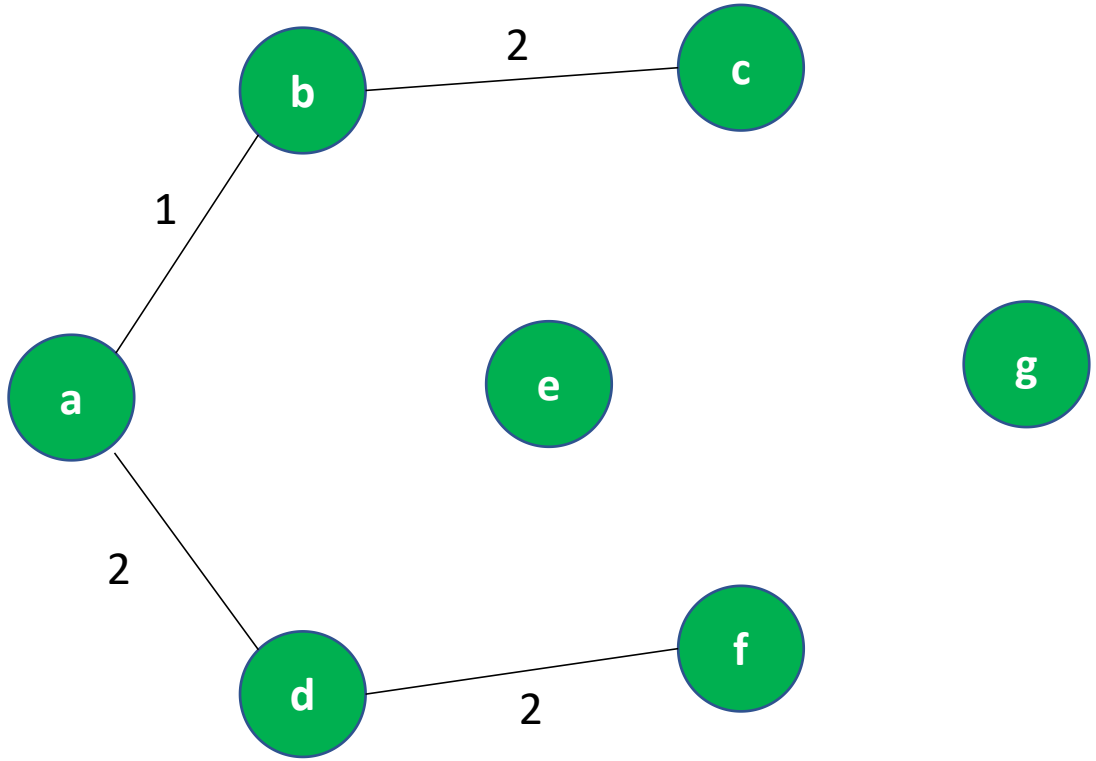


Number of connected components = 4

A = {ab, bc, df }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

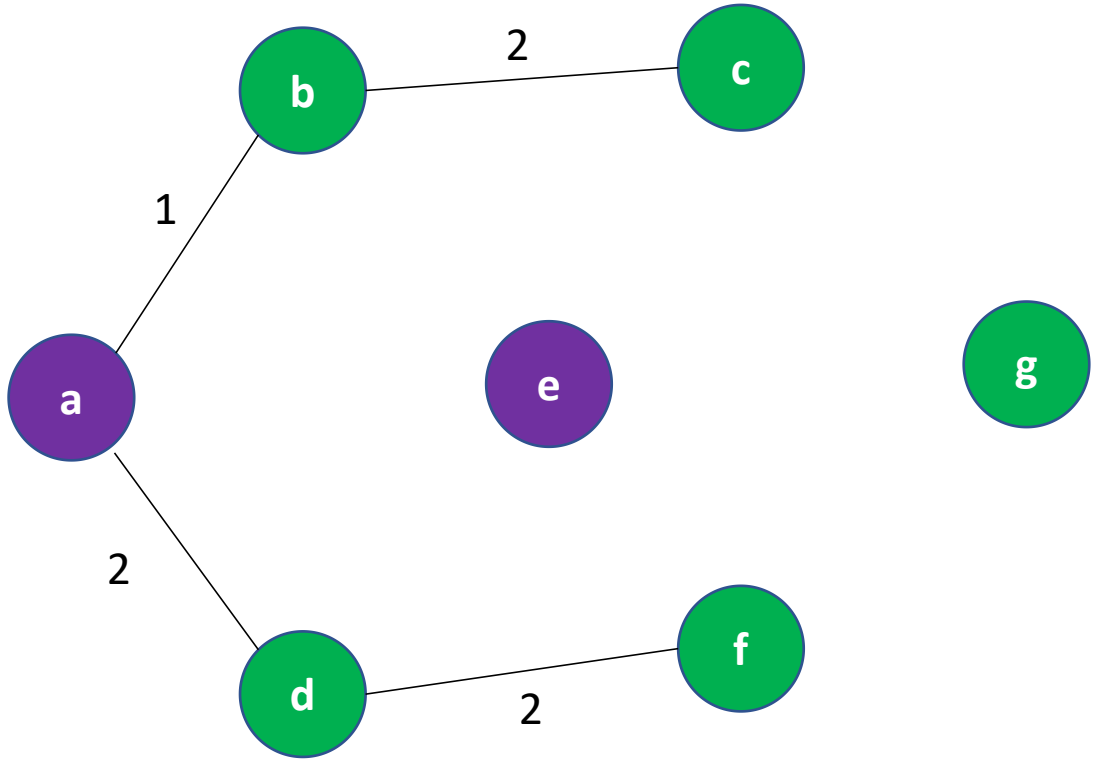


Number of connected components = 3

A = {ab, bc, df, ad }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

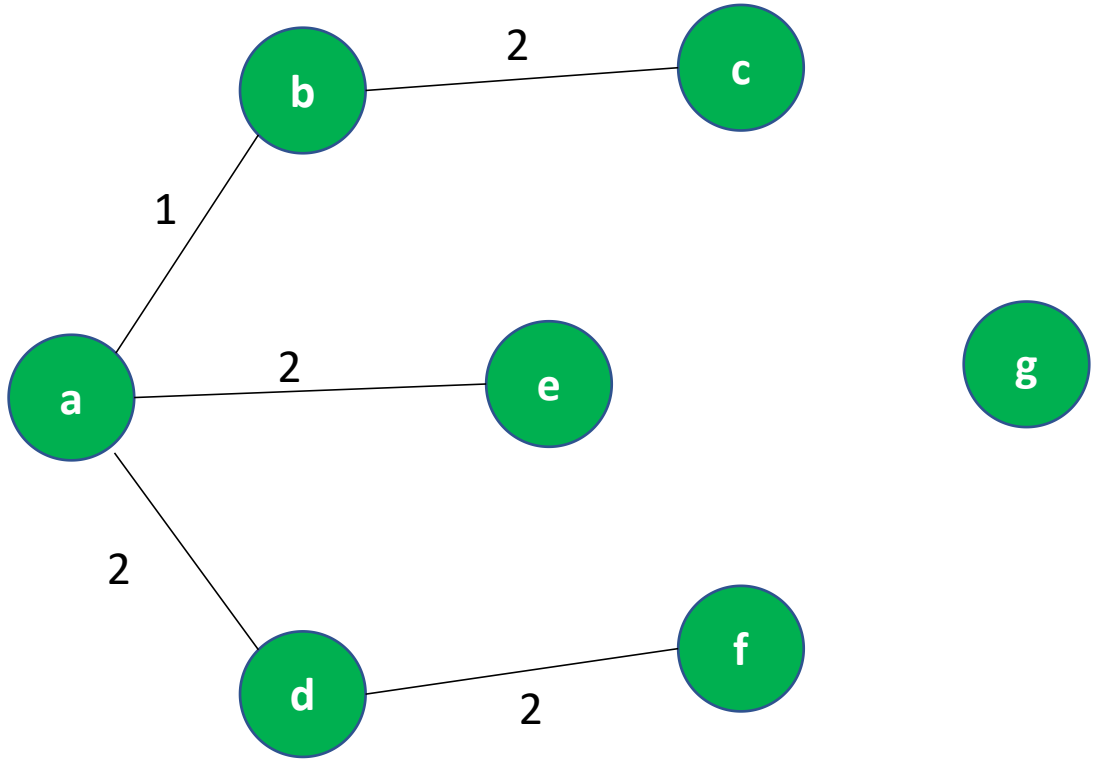


Number of connected components = 3

A = {ab, bc, df, ad }

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

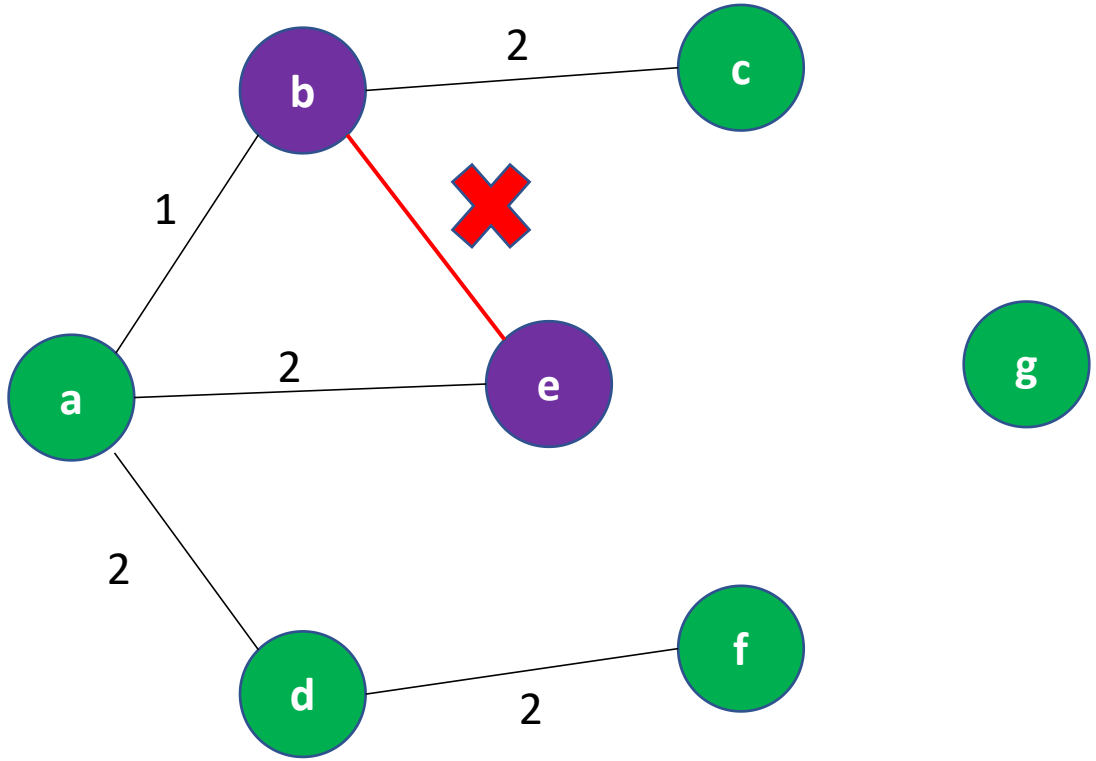


Number of connected components = 2

A = {ab, bc, df, ad, ae}

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

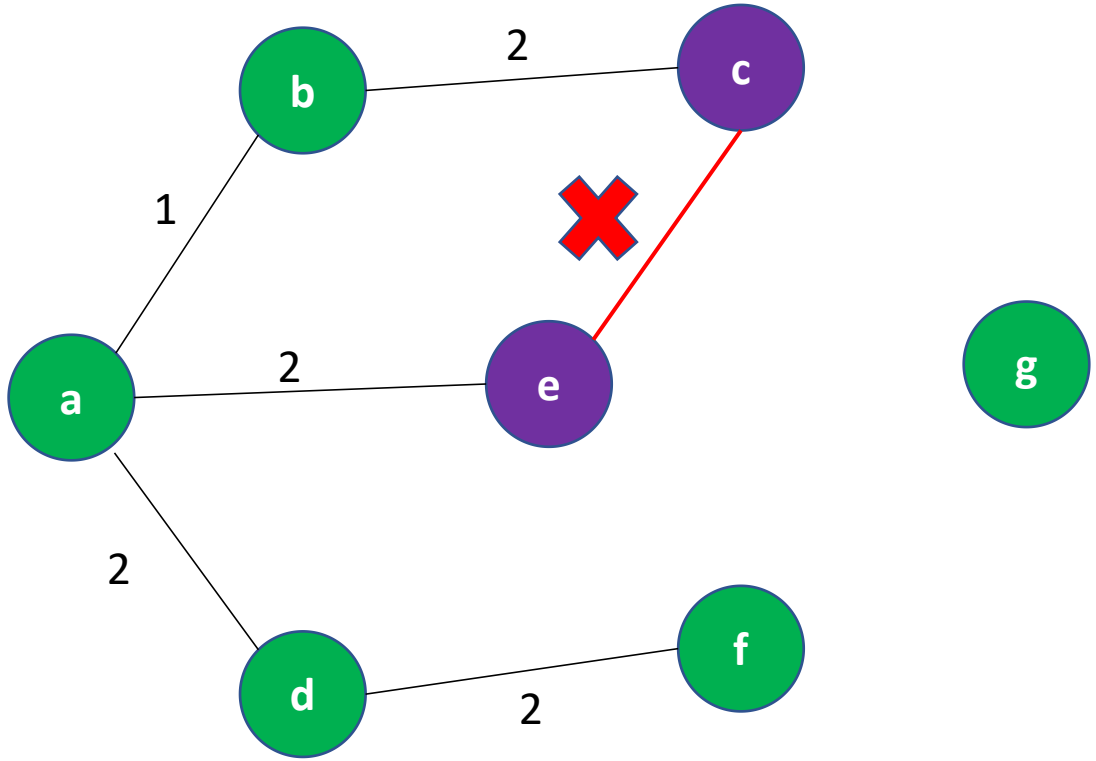


Number of connected components = 2

A = {ab, bc, df, ad, ae}

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

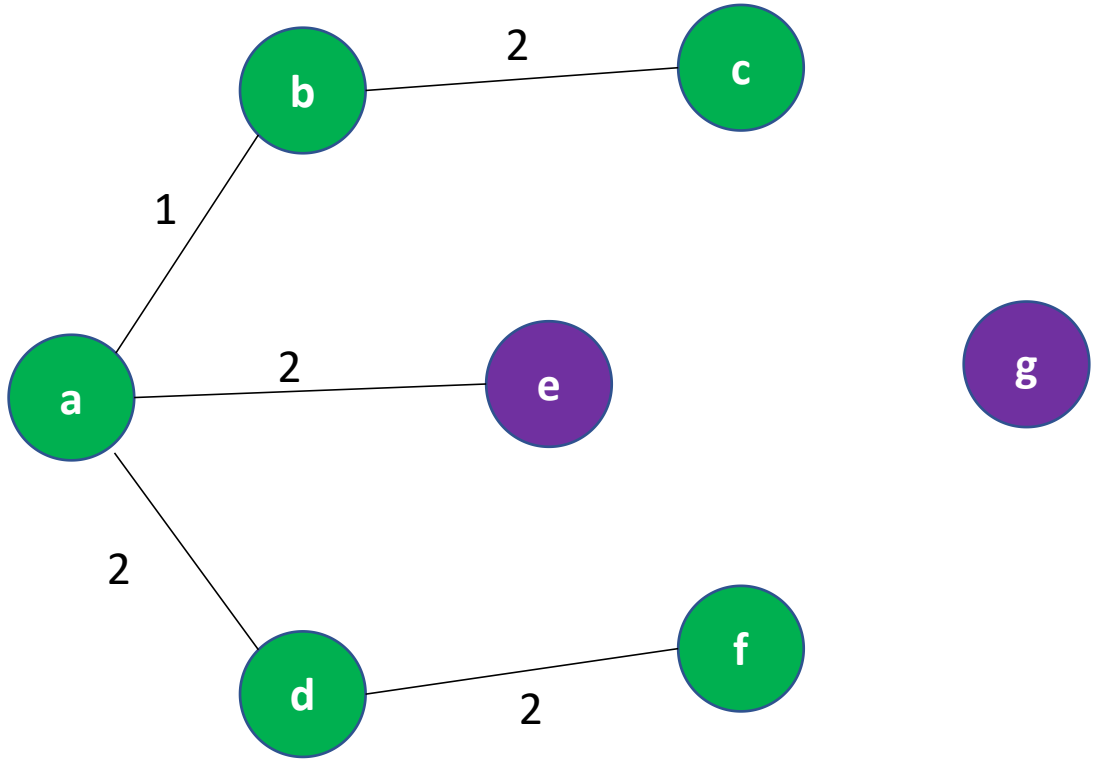


Number of connected components = 2

A = {ab, bc, df, ad, ae}

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

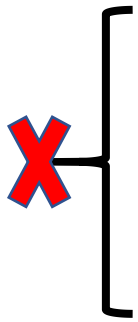


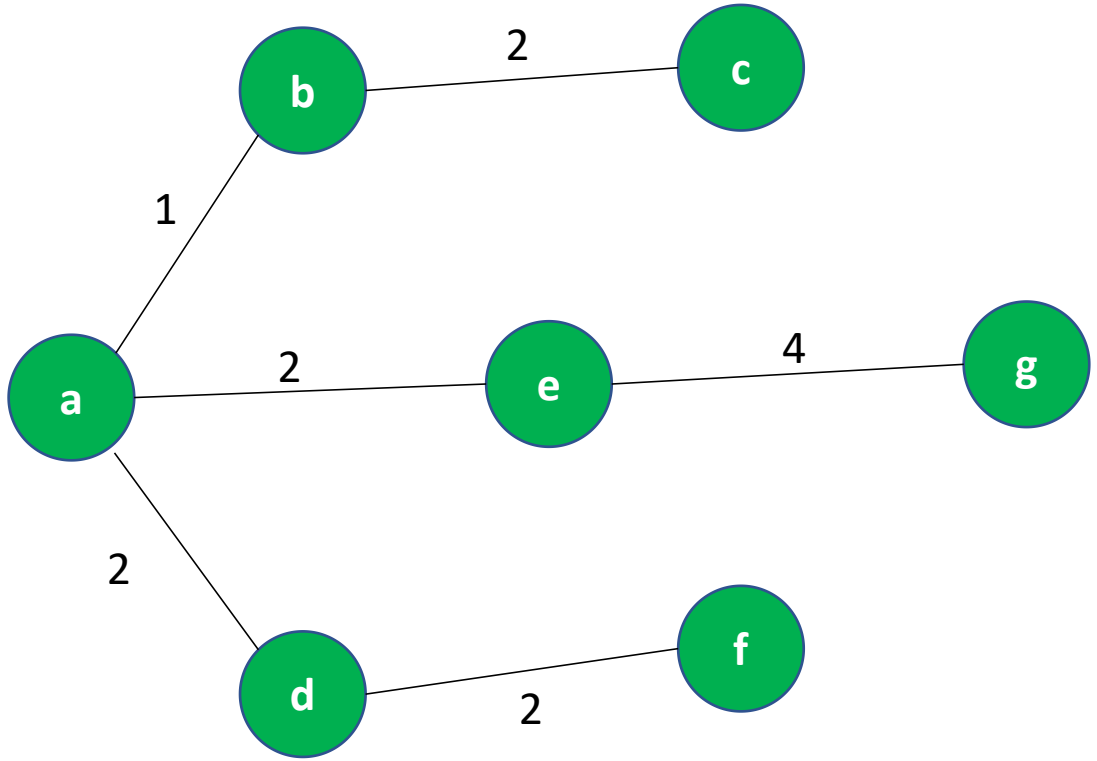
Number of connected components = 2

A = {ab, bc, df, ad, ae}

Kruskal's Algorithm

Edge	Weight
ab	1
bc	2
df	2
ad	2
ae	2
be	3
ce	3
eg	4
ef	5
fg	5
de	6
cg	6

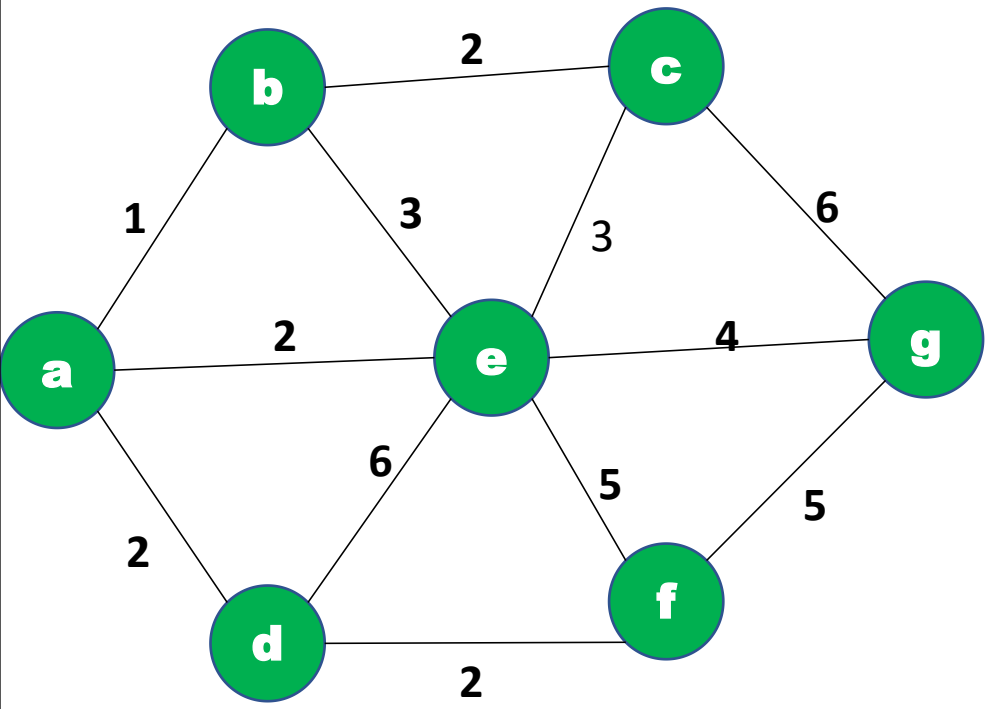




Number of connected components = 1

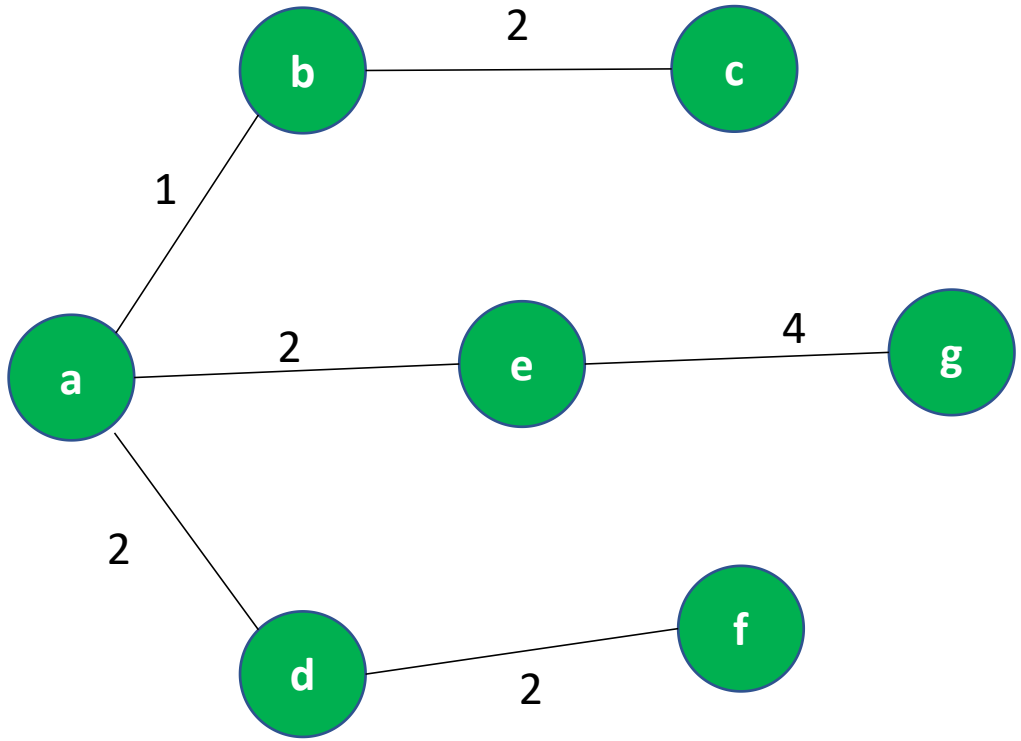
A = {ab, bc, df, ad, ae, eg}

Kruskal's Algorithm



A = {ab, bc, df, ad, ae, eg}

MST



Cost = 1 + 2 + 2 + 4 + 2 + 2 = 13

Performance Analysis of Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

O(1)

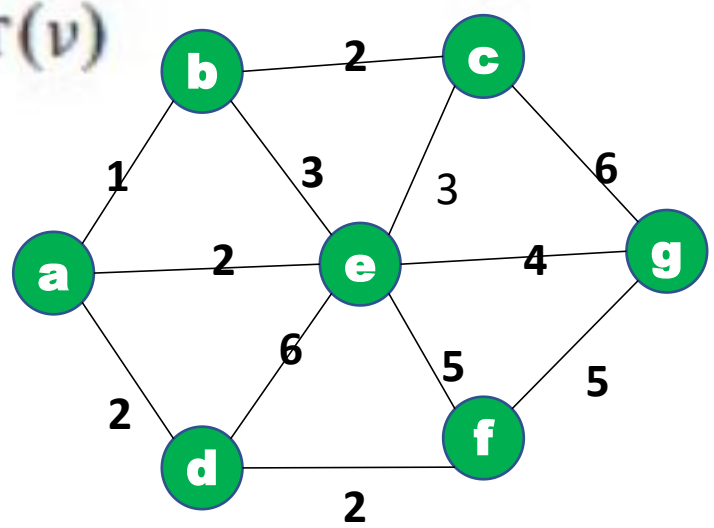
O(V)

O(ElogE)

O(V+E)

=

O(ElogE)



Performance Analysis of Kruskal's Algorithm

Time Complexity of Kruskal's Algorithm = O(ElogE)

In Dense Graph $E = O(V^2)$ hence

Time Complexity of Kruskal's Algorithm $= O(E \log V^2)$
$= O(E * 2 \log V)$
$= O(E * \log V)$

Comparison b/w Prims and Kruskal's Algorithm

Prim's Algorithm

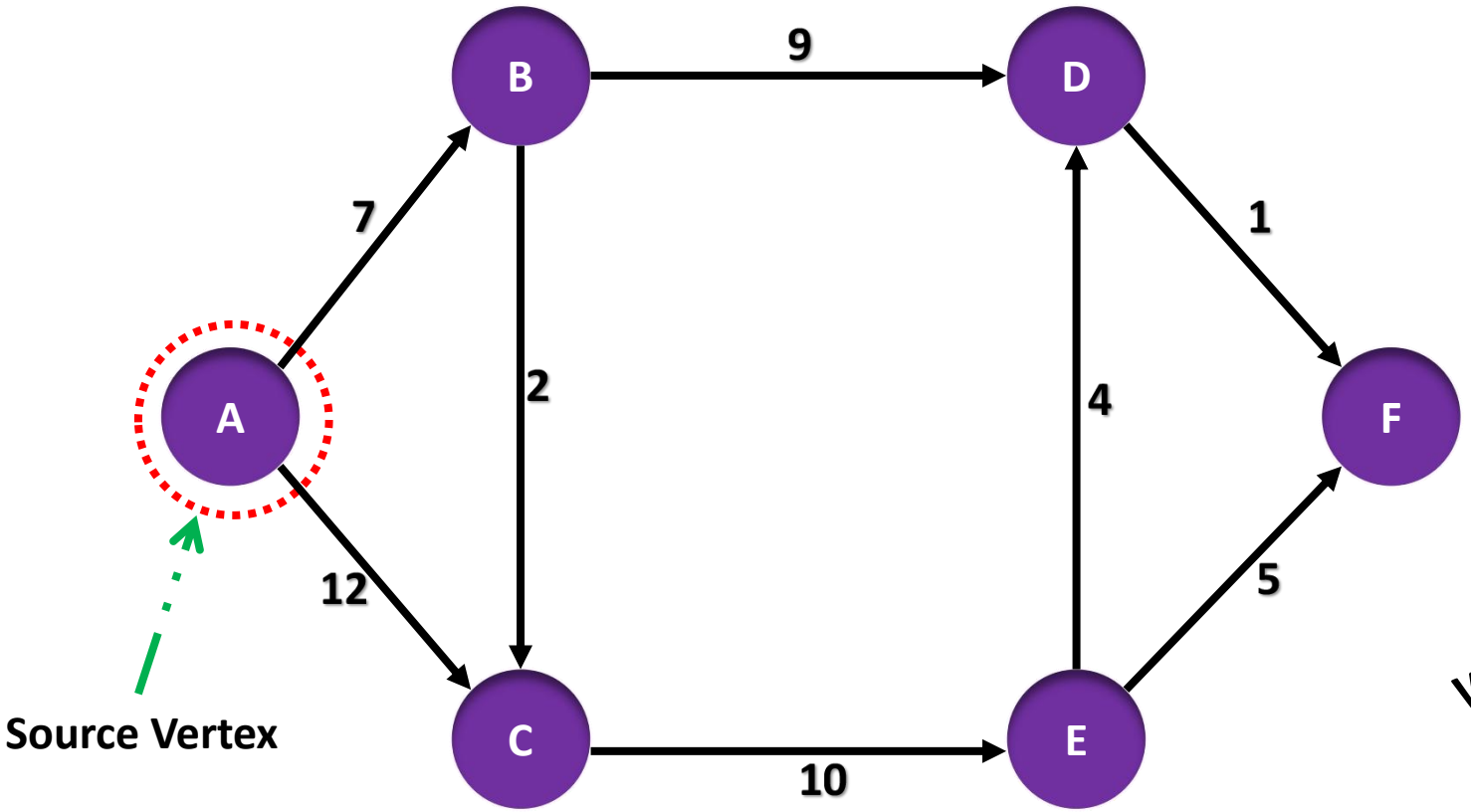
1. In prim's algorithm the tree we are making or growing always remain connected
2. It starts to build MST from any arbitrary node
3. Prim's algorithm is faster for dense graph
4. Adjacency matrix, Binary heap and Fibonacci heap is used for implementation

Kruskal's Algorithm

1. In Kruskal's algorithm the tree we are making or growing may be disconnected
2. It starts building the MST by selecting minimum weight edge.
3. Kruskal's algorithm is faster for sparse graph
4. Disjoint set data structure is used for implementation

Single Source Shortest Path Problem

Context: Directed(or Undirected) Weighted Graph



GOAL

We are interested in finding shortest path and it's cost from A to all vertices of graph.

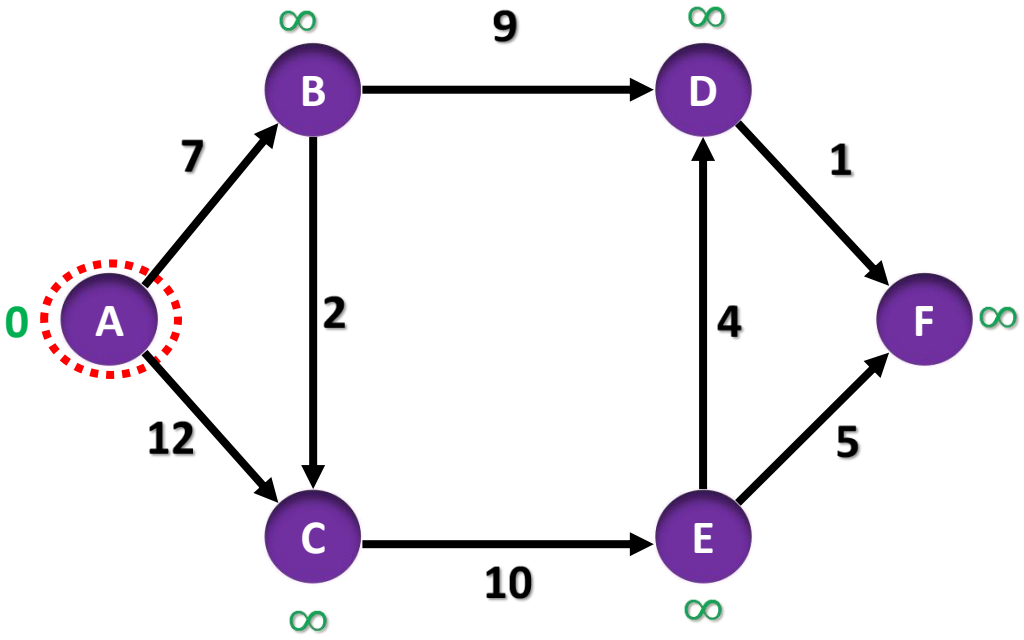
Single Source Shortest Path Problem

Solution: Dijkstra's Algorithm (Got TURING award in 1972)

Interesting facts about Dijkstra :

https://www.youtube.com/watch?v=pxWjlLDRoAs

Dijkstra's Algorithm



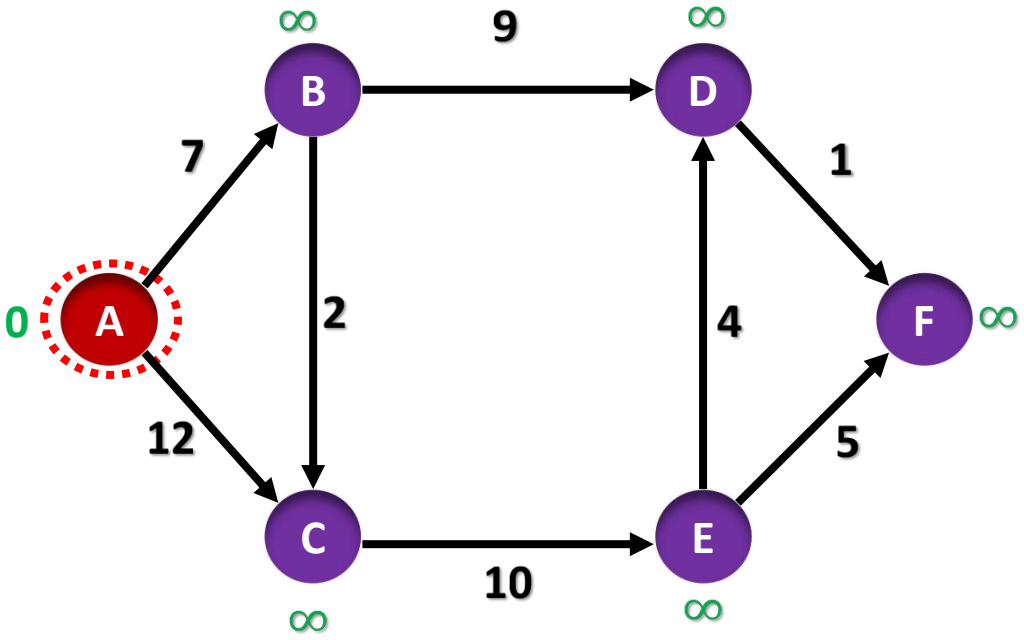
	A	B	C	D	E	F
d[v]	0	∞	∞	∞	∞	∞
$\pi[v]$	Nil	Nil	Nil	Nil	Nil	Nil

$Q = \langle A^0, B^\infty, C^\infty, D^\infty, E^\infty, F^\infty \rangle$

$S = \{\ \}$

A = ExtractMin(Q)

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	∞	∞	∞	∞	∞
$\pi[v]$	Nil	Nil	Nil	Nil	Nil	Nil

$Q = \langle B^{\infty}, C^{\infty}, D^{\infty}, E^{\infty}, F^{\infty} \rangle$

$S = \{ A^{0} \}$

For AB : $d[A] + w(A, B) < d[B]$

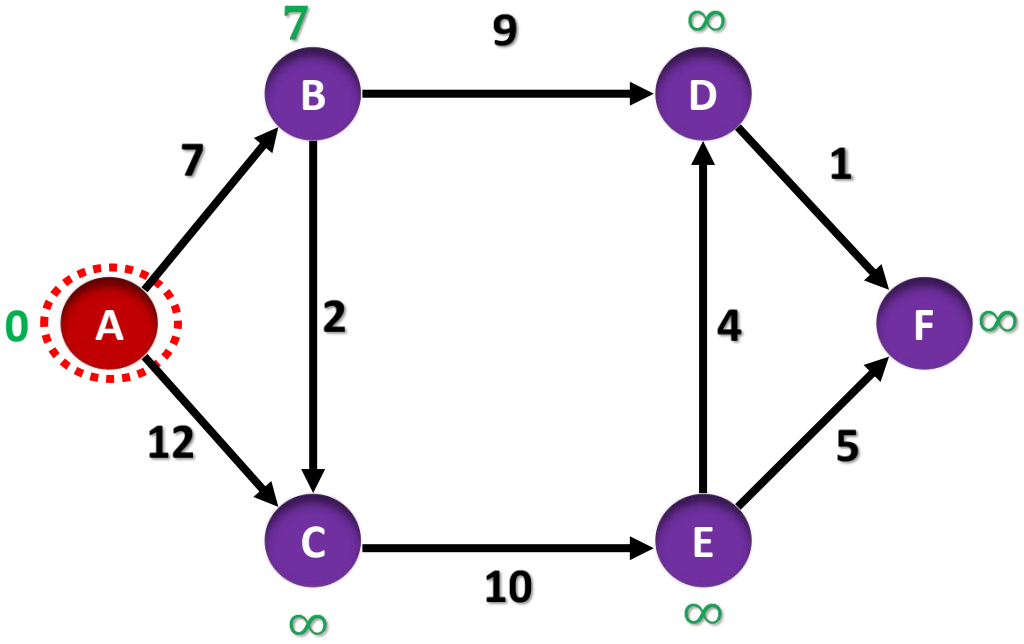
$0 + 7 < \infty$

TRUE

$d[B] = 7$

$\pi[B] = A$

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	∞	∞	∞	∞
$\pi[v]$	Nil	A	Nil	Nil	Nil	Nil

$Q = \langle B^{7}, C^{\infty}, D^{\infty}, E^{\infty}, F^{\infty} \rangle$

$S = \{ A^{0} \}$

For AC : d[A] + w(A, C) < d[C]

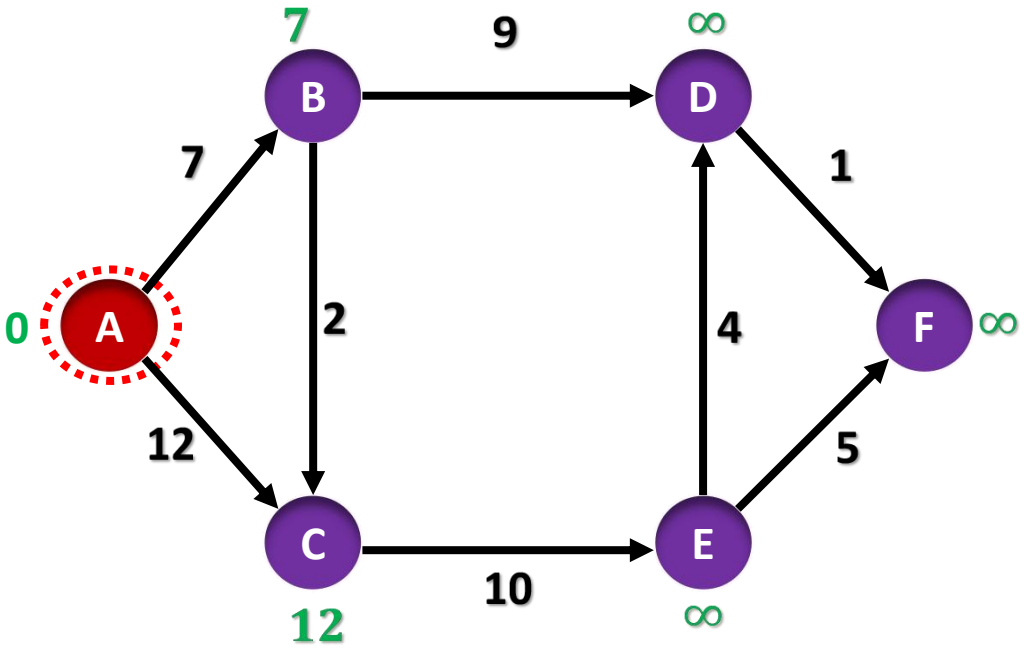
0 + 12 < ∞

} TRUE

d[C] = 12

$\pi[C] = A$

Dijkstra's Algorithm



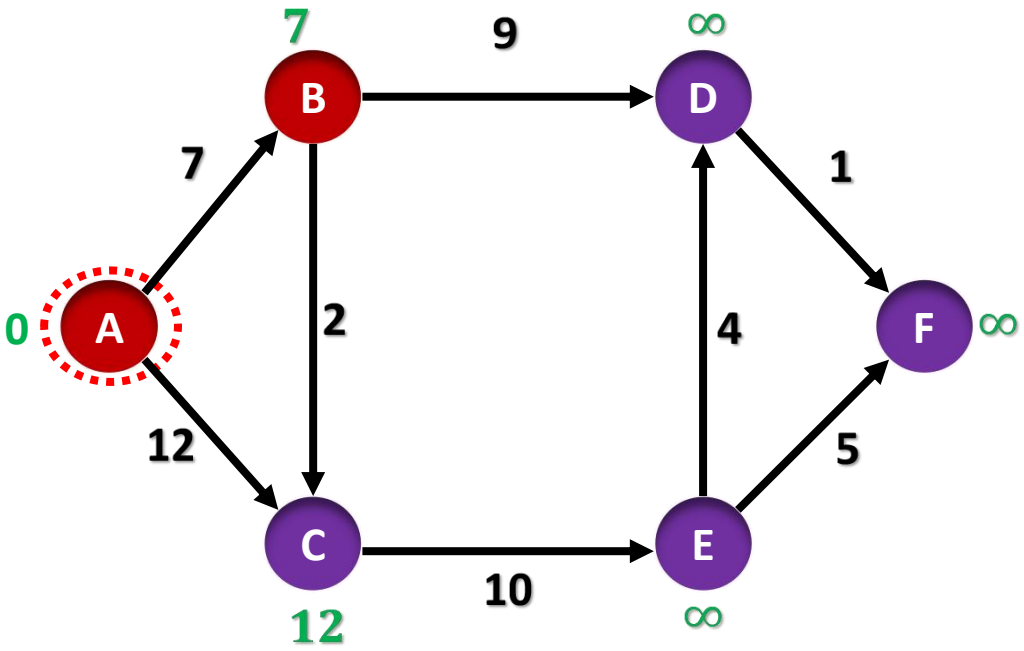
	A	B	C	D	E	F
d[v]	0	7	12	∞	∞	∞
$\pi[v]$	Nil	A	A	Nil	Nil	Nil

$Q = \langle B^{7}, C^{12}, D^{\infty}, E^{\infty}, F^{\infty} \rangle$

$S = \{ A^{0} \}$

ExtractMin(Q)

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	12	∞	∞	∞
$\pi[v]$	Nil	A	A	Nil	Nil	Nil

$Q = < C^{12}, D^{\infty}, E^{\infty}, F^{\infty} >$

$S = \{ A^{0}, B^{7} \}$

For BD : $d[B] + w(B, D) < d[D]$

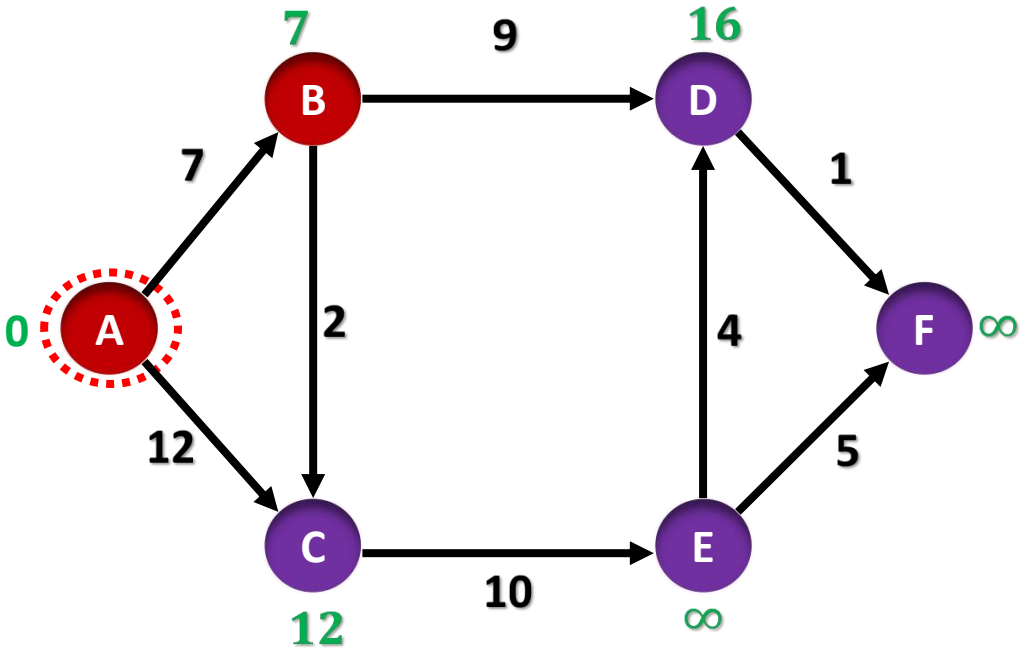
$7 + 9 < \infty$

} TRUE

$d[D] = 16$

$\pi[D] = B$

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	12	16	∞	∞
$\pi[v]$	Nil	A	A	B	Nil	Nil

$Q = < C^{12}, D^{16}, E^{\infty}, F^{\infty} >$

$S = \{ A^{0}, B^{7} \}$

For BD : $d[B] + w(B, D) < d[D]$

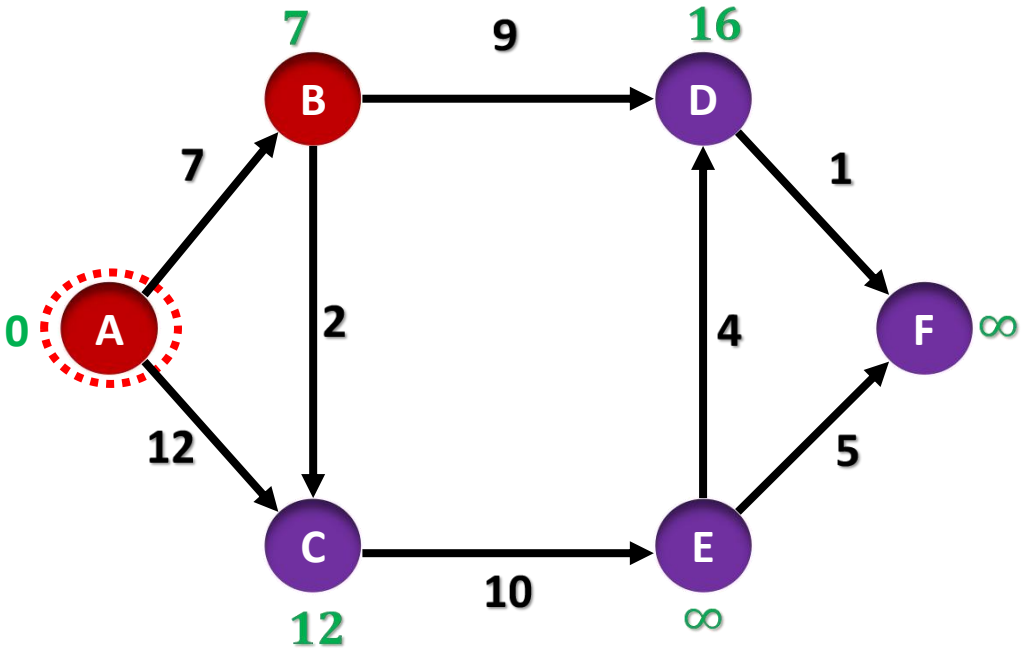
$7 + 9 < \infty$

TRUE

$d[D] = 16$

$\pi[D] = B$

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	12	16	∞	∞
$\pi[v]$	Nil	A	A	B	Nil	Nil

$Q = < C^{12}, D^{16}, E^{\infty}, F^{\infty} >$

$S = \{ A^{0}, B^{7} \}$

For BC : d[B] + w(B, C) < d[C]

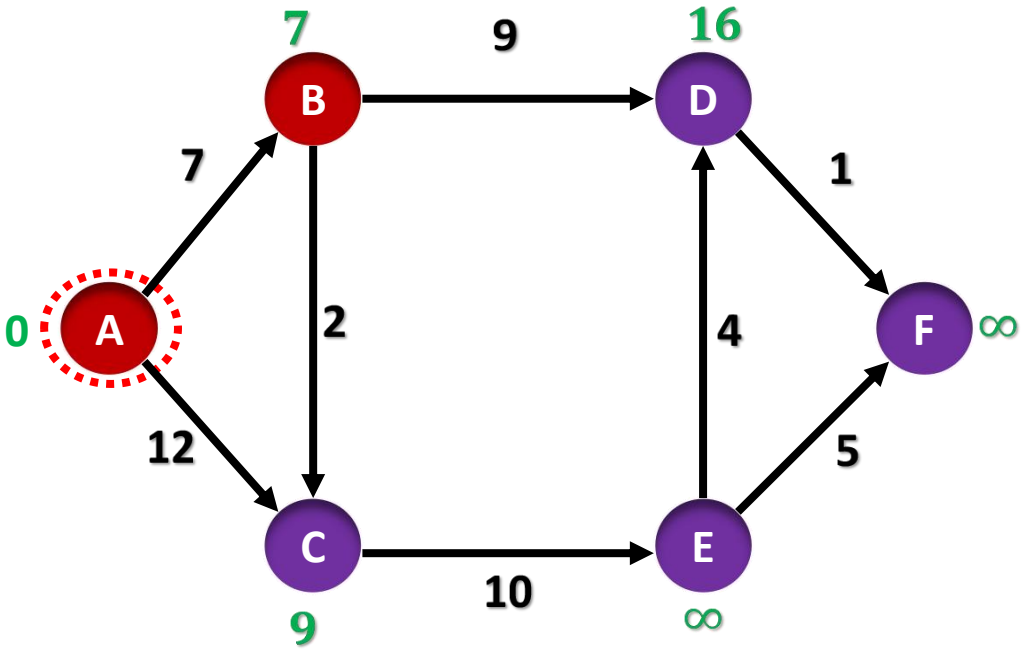
7 + 2 < 12

TRUE

d[C] = 9

$\pi[C] = B$

Dijkstra's Algorithm



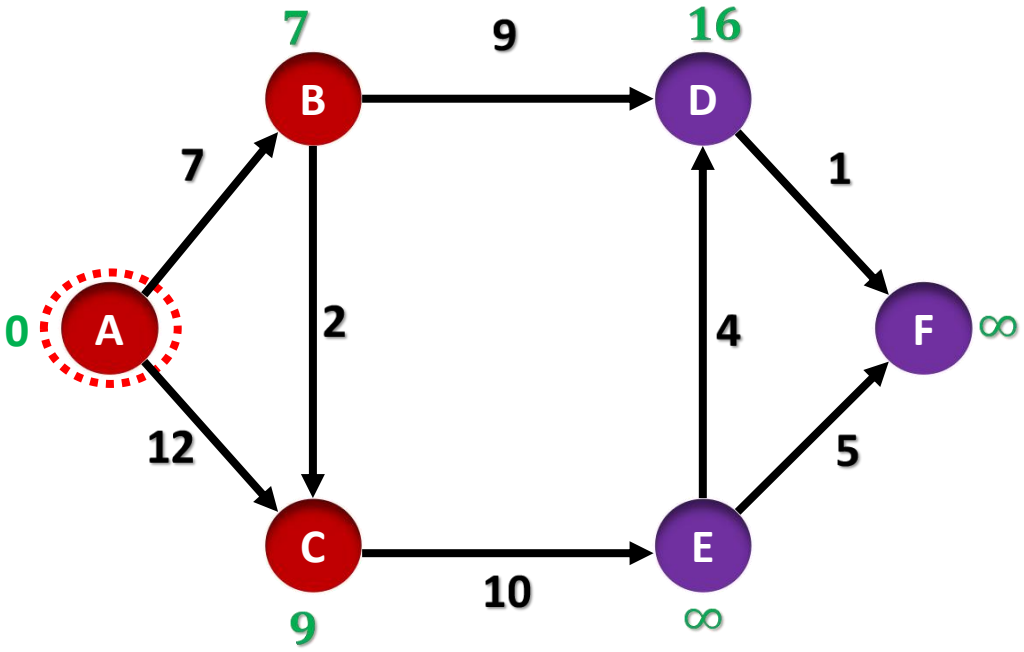
	A	B	C	D	E	F
d[v]	0	7	9	16	∞	∞
$\pi[v]$	Nil	A	B	B	Nil	Nil

$Q = < C^{9}, D^{16}, E^{\infty}, F^{\infty} >$

$S = \{ A^{0}, B^{7} \}$

ExtractMin(Q)

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	9	16	∞	∞
$\pi[v]$	Nil	A	B	B	Nil	Nil

$Q = < D^{16}, E^{\infty}, F^{\infty} >$

$S = \{ A^{0}, B^{7}, C^{9} \}$

For CE : $d[C] + w(C, E) < d[E]$

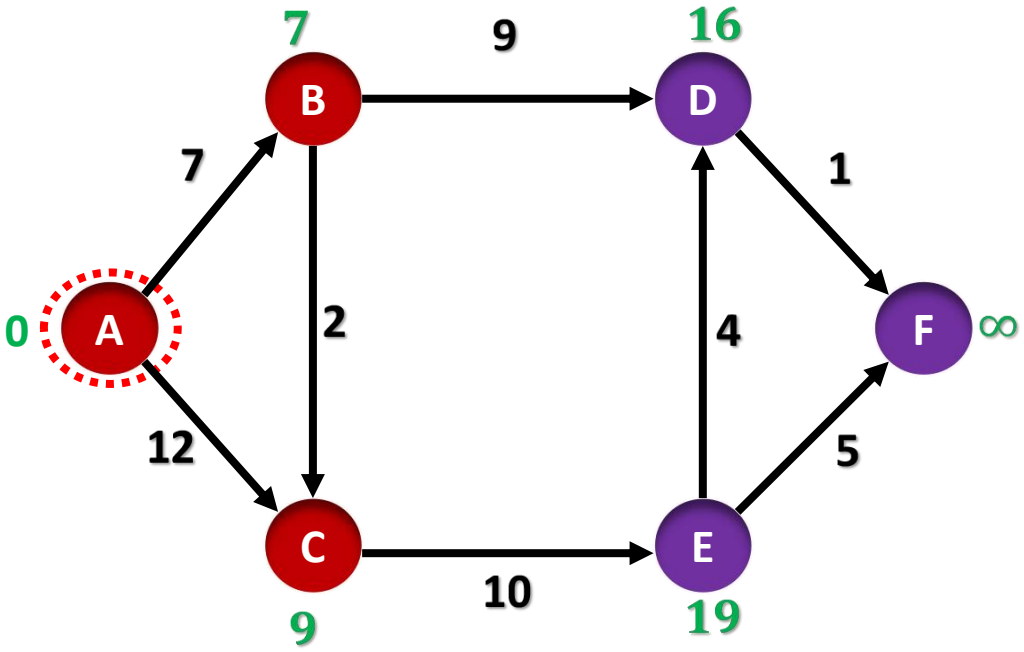
$9 + 10 < \infty$

TRUE

$d[E] = 19$

$\pi[E] = C$

Dijkstra's Algorithm



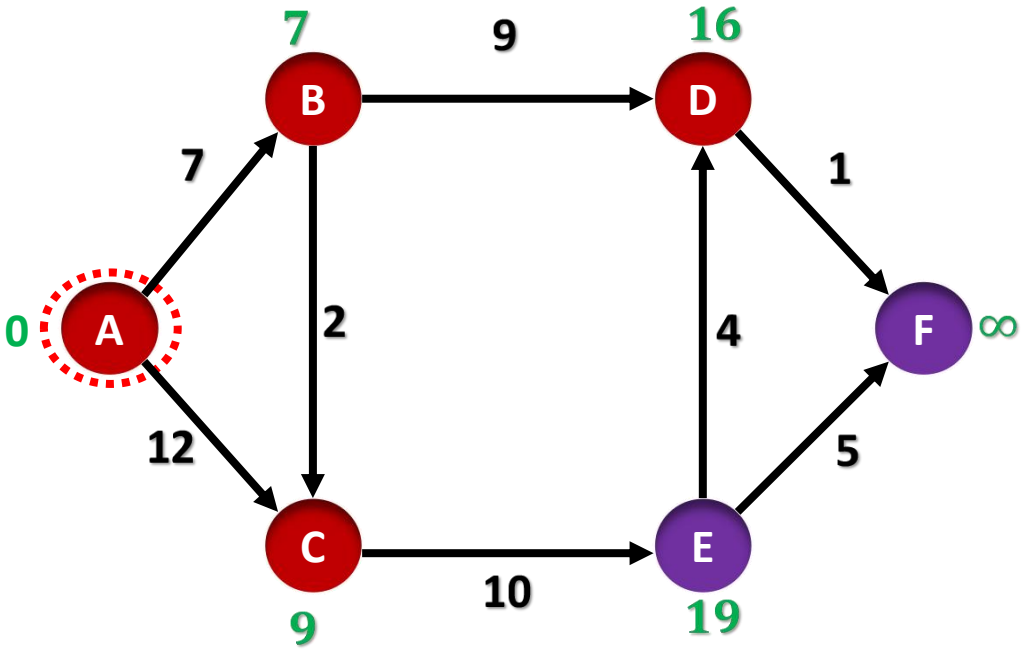
	A	B	C	D	E	F
d[v]	0	7	9	16	19	∞
$\pi[v]$	Nil	A	B	B	C	Nil

$Q = < D^{16}, E^{19}, F^{\infty} >$

$S = \{ A^{0}, B^{7}, C^{9} \}$

ExtractMin(Q)

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	9	16	19	∞
$\pi[v]$	Nil	A	B	B	C	Nil

$Q = < E^{19}, F^{\infty} >$

$S = \{ A^{0}, B^{7}, C^{9}, D^{16} \}$

For DF : $d[D] + w(D, F) < d[F]$

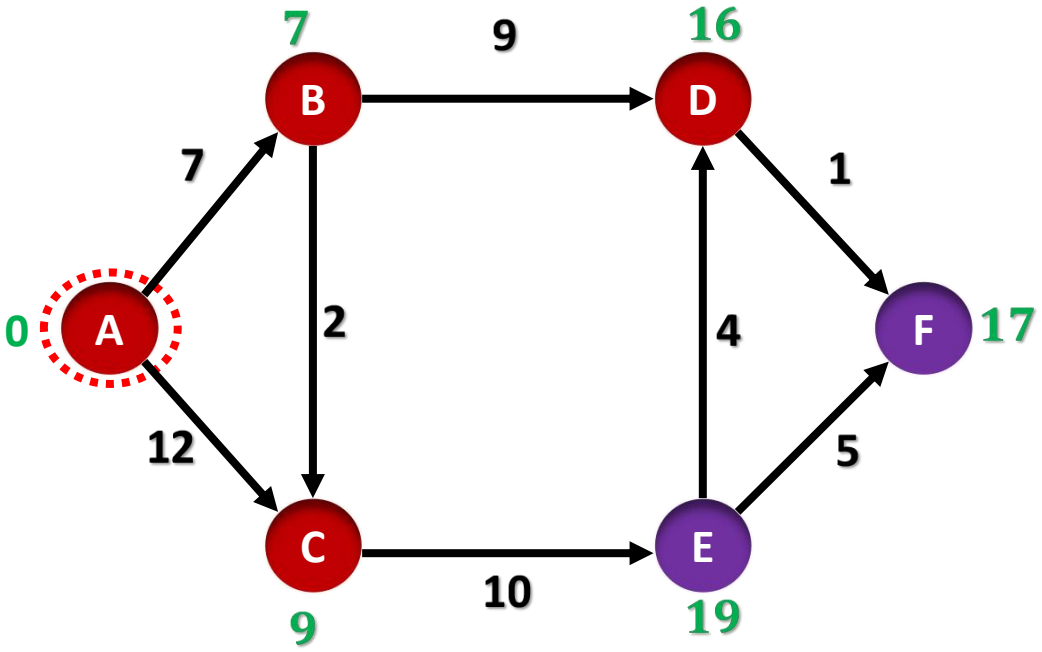
$16 + 1 < \infty$

TRUE

d[F] = 17

$\pi[F] = D$

Dijkstra's Algorithm



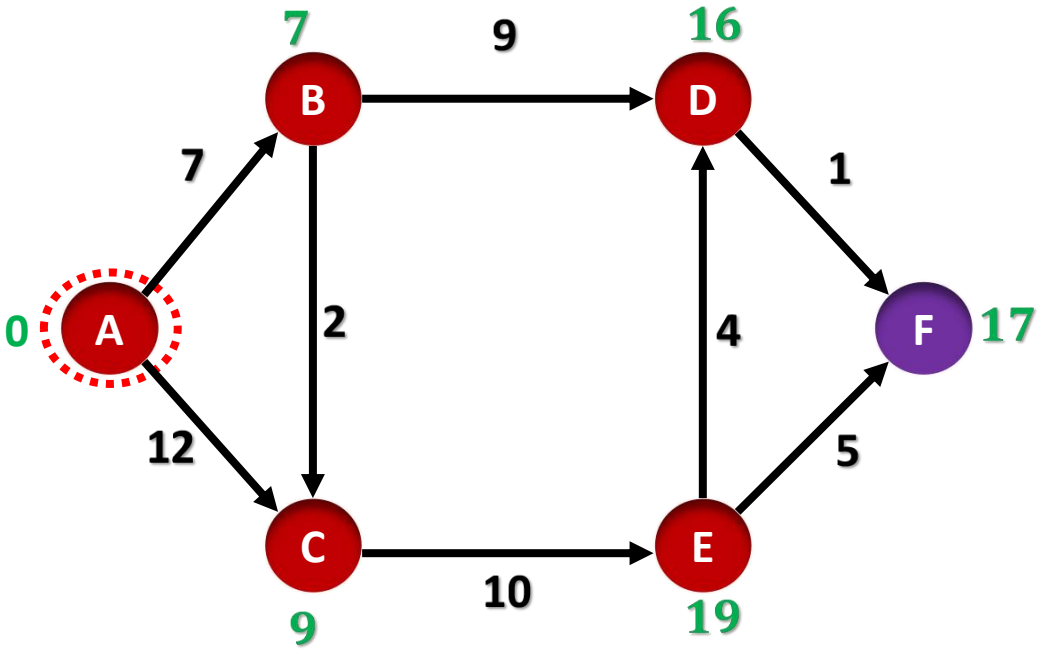
	A	B	C	D	E	F
d[v]	0	7	9	16	19	17
$\pi[v]$	Nil	A	B	B	C	D

$Q = < E^{19}, F^{17} >$

$S = \{ A^0, B^7, C^9, D^{16} \}$

ExtractMin(Q)

Dijkstra's Algorithm



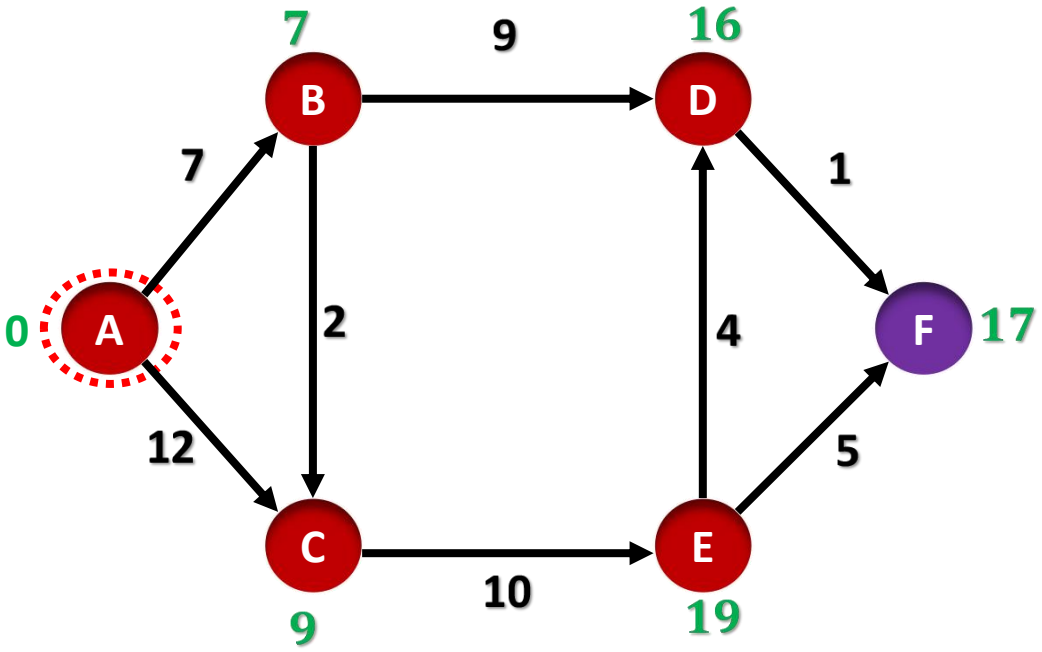
	A	B	C	D	E	F
d[v]	0	7	9	16	19	17
$\pi[v]$	Nil	A	B	B	C	D

$Q = < E^{19} >$

$S = \{ A^0, B^7, C^9, D^{16}, F^{17} \}$

No Adjacent Vertex of F

Dijkstra's Algorithm



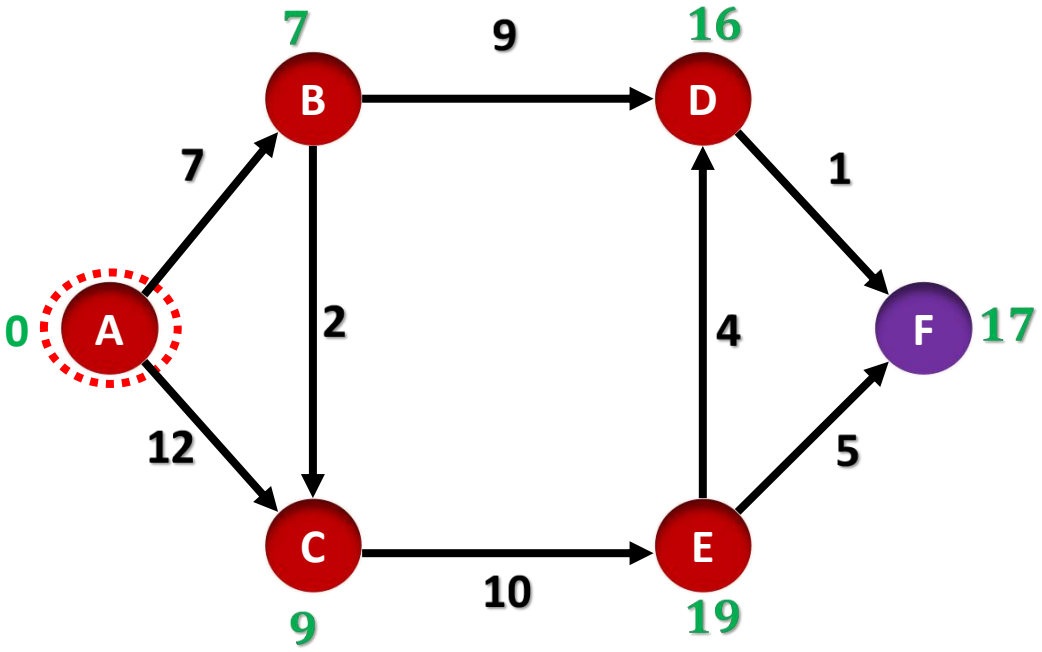
	A	B	C	D	E	F
d[v]	0	7	9	16	19	17
$\pi[v]$	Nil	A	B	B	C	D

$Q = < E^{19} >$

$S = \{ A^0, B^7, C^9, D^{16}, F^{17} \}$

ExtractMin(Q)

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	9	16	19	17
$\pi[v]$	Nil	A	B	B	C	D

Q = < >

S = { A^0, B^7, C^9, D^{16}, F^{17}, E^{19} }

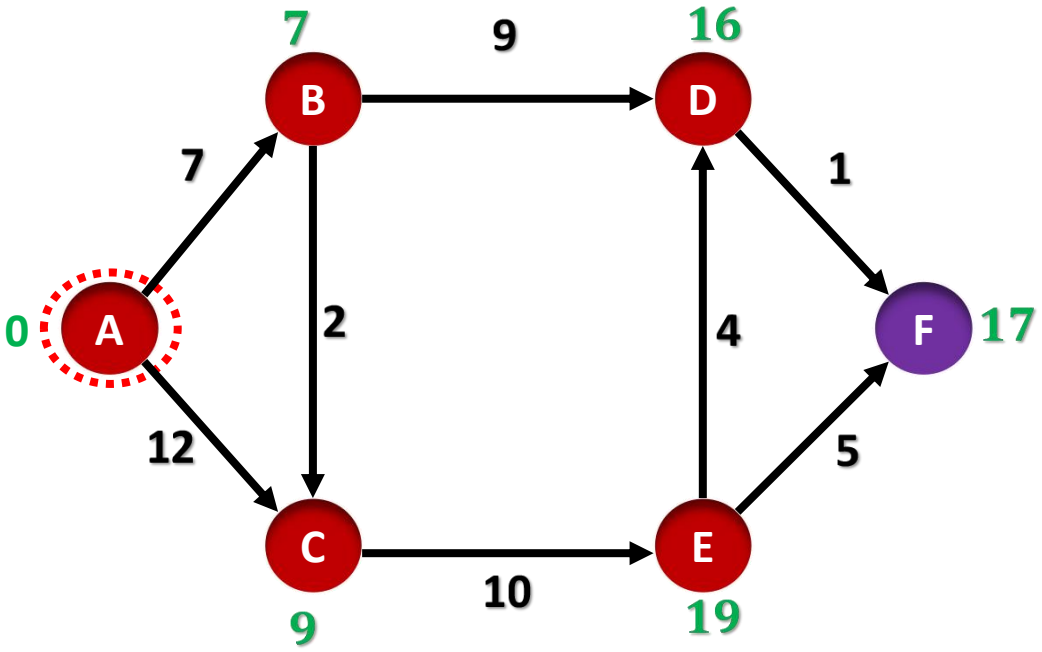
For ED : d[E] + w(E, D) < d[D]

19 + 4 < 16

FALSE

No Update

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	9	16	19	17
$\pi[v]$	Nil	A	B	B	C	D

Q = < >

S = { A^0, B^7, C^9, D^{16}, F^{17} , E^{19}}

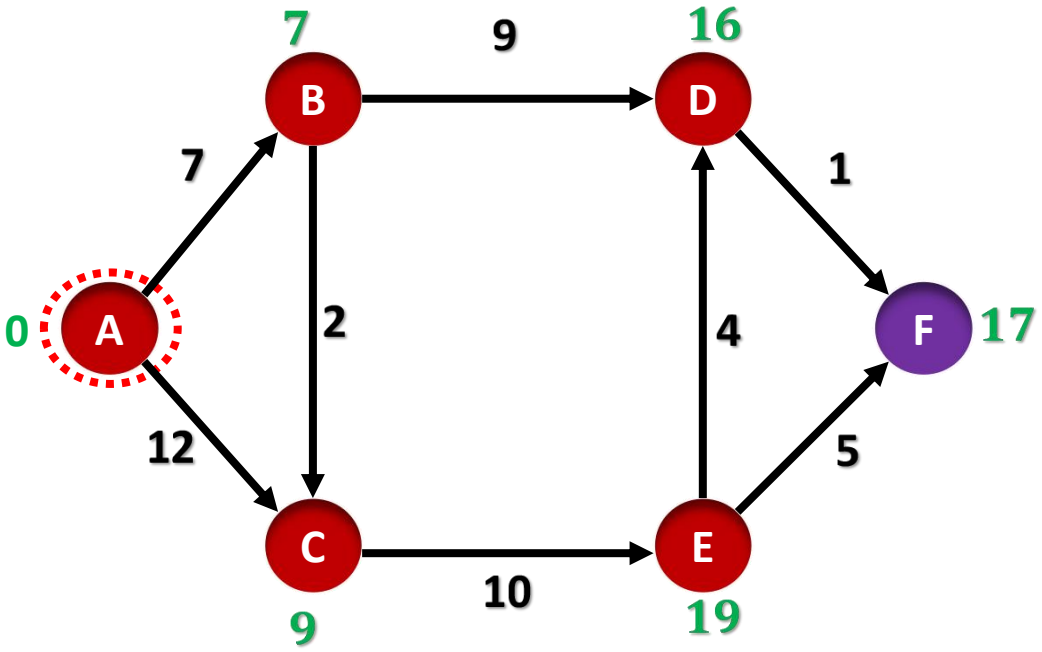
For EF : d[E] + w(E, F) < d[F]

19 + 5 < 17

FALSE

No Update

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	9	16	19	17
$\pi[v]$	Nil	A	B	B	C	D

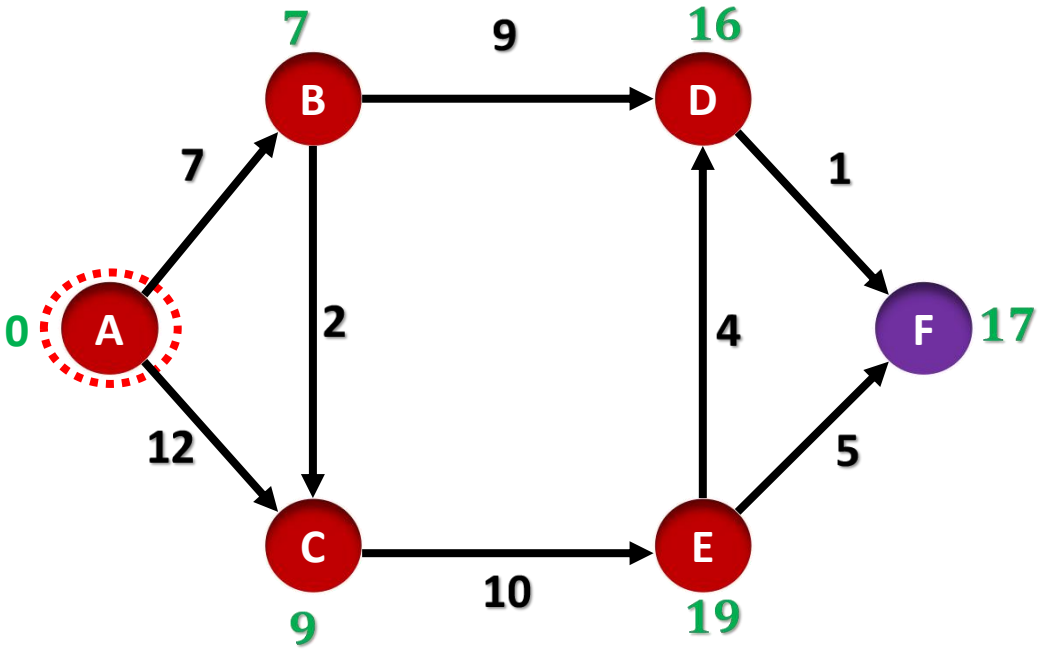
Q = < >

S = { A^0, B^7, C^9, D^{16}, E^{19}, F^{17}}

There is NO edge to relax
&
Queue Q is Empty

→ Dijkstra's Algo Terminates

Dijkstra's Algorithm



	A	B	C	D	E	F
d[v]	0	7	9	16	19	17
$\pi[v]$	Nil	A	B	B	C	D

$S = \{ A^0, B^7, C^9, D^{16}, E^{19}, F^{17} \}$

Shortest Path from A to F : A → B → D → F

Shortest Path from A to E : A → B → C → E

Priority Queue: Heap

Time Complexity

Operation	Cost
Build-Heap	O(n)
Extract-Min/Delete	O(logn)
Update/Relax	O(logn)
Insert	O(logn)

Dijkstra's Algorithm: Cost Analysis

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Cost of ExtractMin(Q) = O(logV)
# of ExtractMin(Q) = V

Total Cost = O(VlogV)

Dijkstra's Algorithm: Cost Analysis

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Cost of Relax = O(logV)
# of Relax() = E

Total Cost = O(ElogV)

Dijkstra's Algorithm: Cost Analysis

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Cost of ExtractMin(Q) = O(logV)
# of ExtractMin(Q) = V

Total Cost = O(VlogV)

Cost of Relax() = O(logV)
# of Relax() = E

Total Cost = O(ElogV)

Cost = O(VlogV) + O(ElogV)

Cost = O((V + E)LogV)

Dijkstra's Algorithm: Cost Analysis

Time Complexity = $O((V + E)\log V)$

$= O(E \log V)$

Dijkstra's Algorithm

- ✓ Dijkstra's Algorithm fails on negative weighted edges.....(always/not always)?? : not always
- ✓ Dijkstra's Algorithm fails on negative weighted cycle.....(always/not always)?? : always

Bellman Ford Algorithm

```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL

    distance[S] <- 0

    for each vertex V in G

        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V}
            Error: Negative Cycle Exists

    return distance[], previous[]
```

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q

    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    return distance[], previous[]
```

Complexity = O(VE)