More on Greedy Algorithms

Murat Osmanoglu

MST

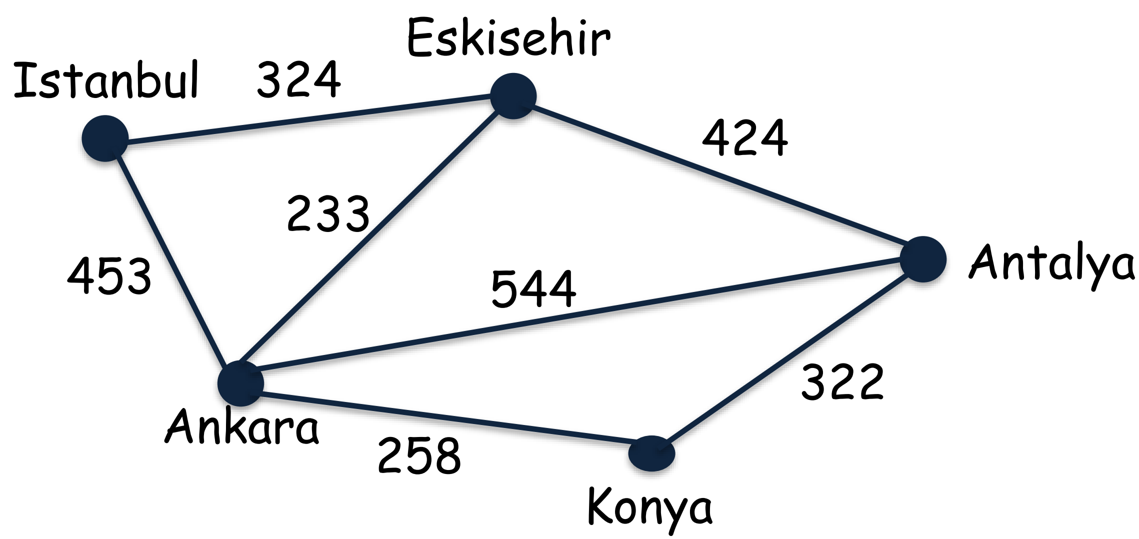
$$G = (V, E)$$

$w : E \rightarrow R$, that assigns a weight to each edge

MST

$$G = (V, E)$$

$w : E \rightarrow R$, that assigns a weight to each edge



MST

- given a connected undirected graph $G = (V,E)$ with real-valued edge weight $w(e)$, an minimum spanning tree (MST) is a subset of the edges $T \subseteq E$ such that T is a tree that connects all vertices of the graphs, and T has minimum total weight.

MST

- given a connected undirected graph $G = (V,E)$ with real-valued edge weight $w(e)$, an minimum spanning tree (MST) is a subset of the edges $T \subseteq E$ such that T is a tree that connects all vertices of the graphs, and T has minimum total weight.

$$w(T) = \Sigma_{e \text{ in } T}\, w(e)$$

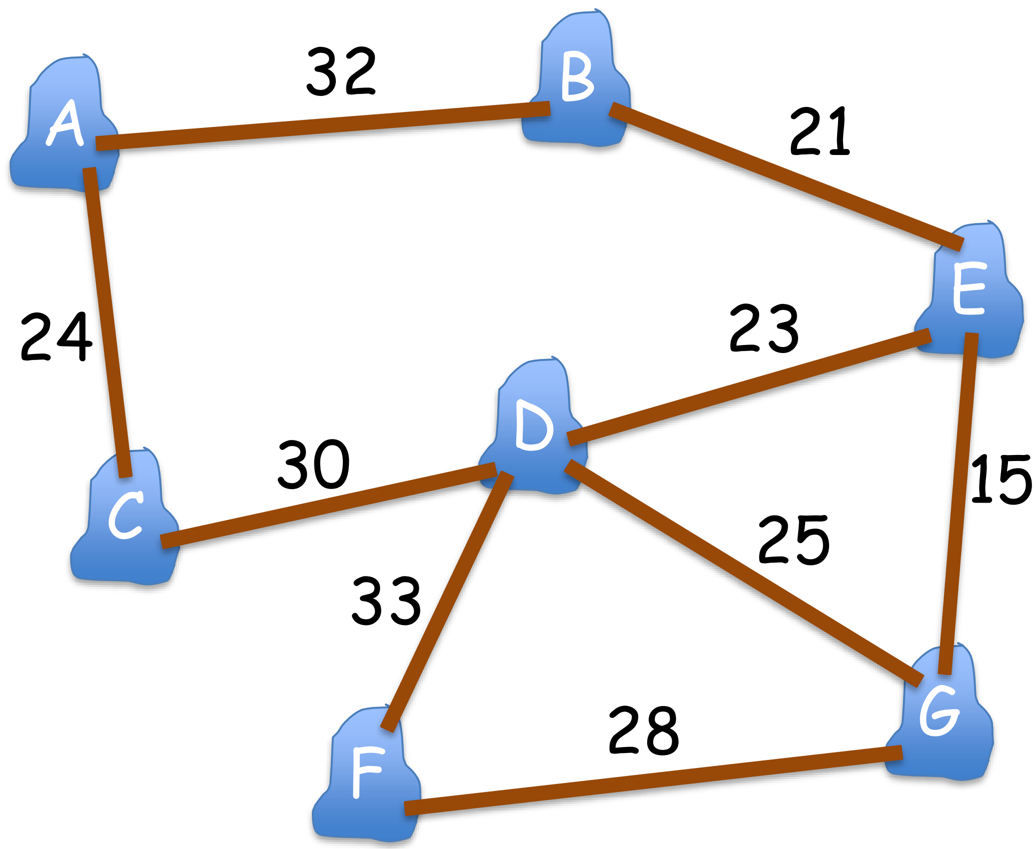
MST

- given a connected undirected graph $G = (V,E)$ with real-valued edge weight $w(e)$, an minimum spanning tree (MST) is a subset of the edges $T \subseteq E$ such that T is a tree that connects all vertices of the graphs, and T has minimum total weight.

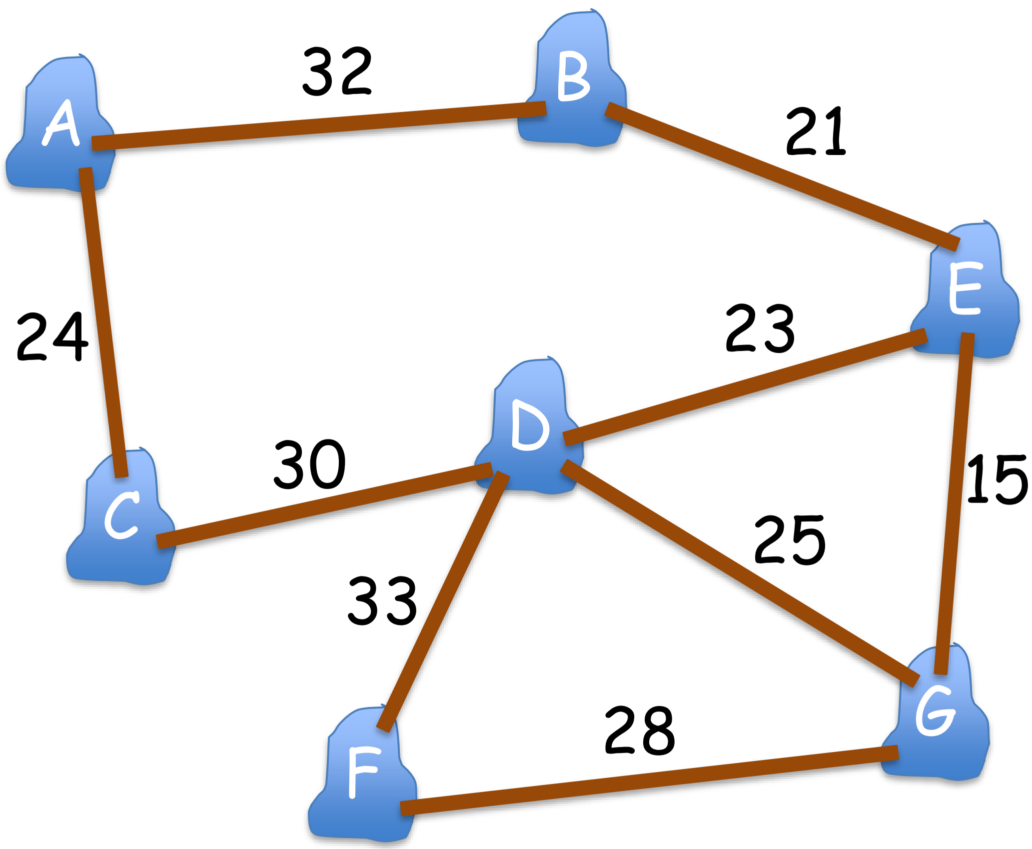
$$w(T) = \Sigma_{e \text{ in } T}\, w(e)$$

- connecting all computers in an office buildings using least amount of cables

MST

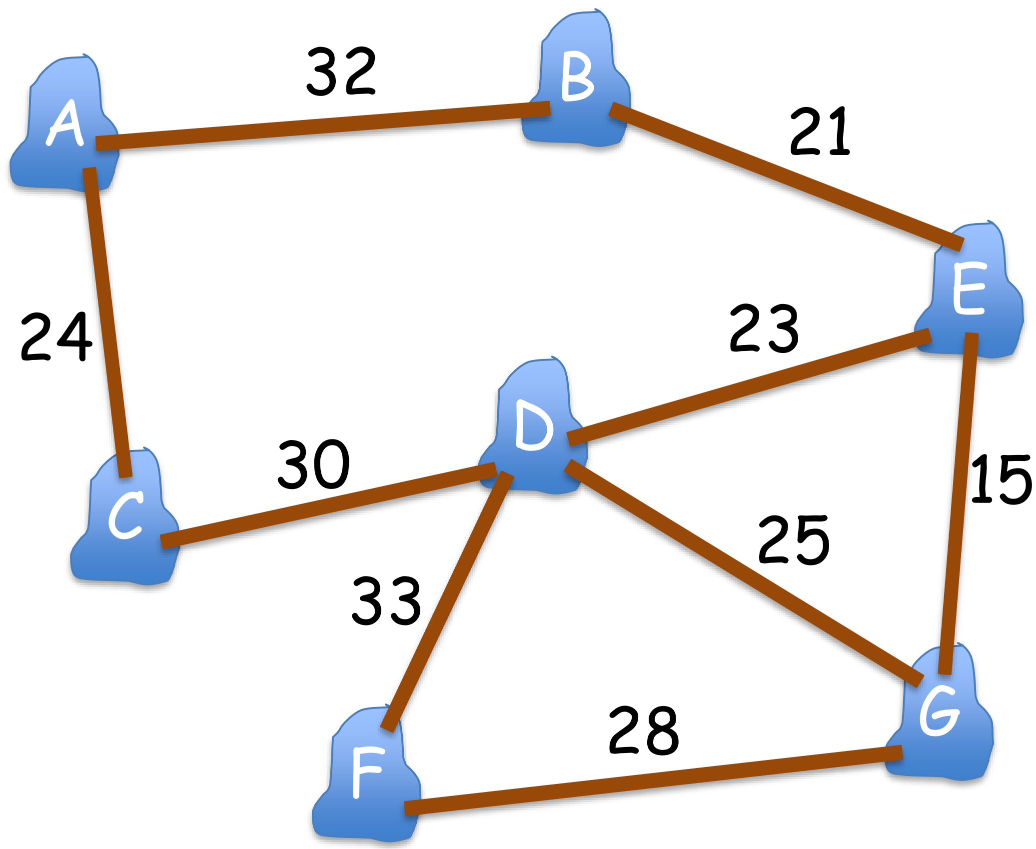


MST



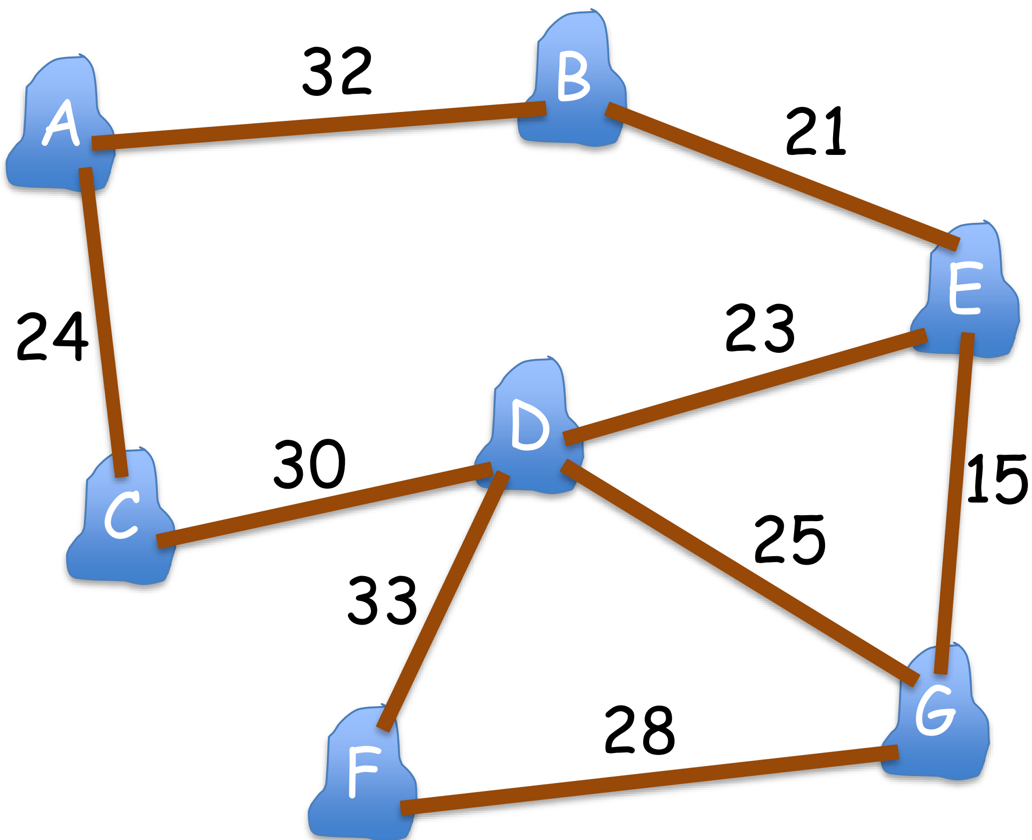
- start with an empty set of edges A

MST



- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

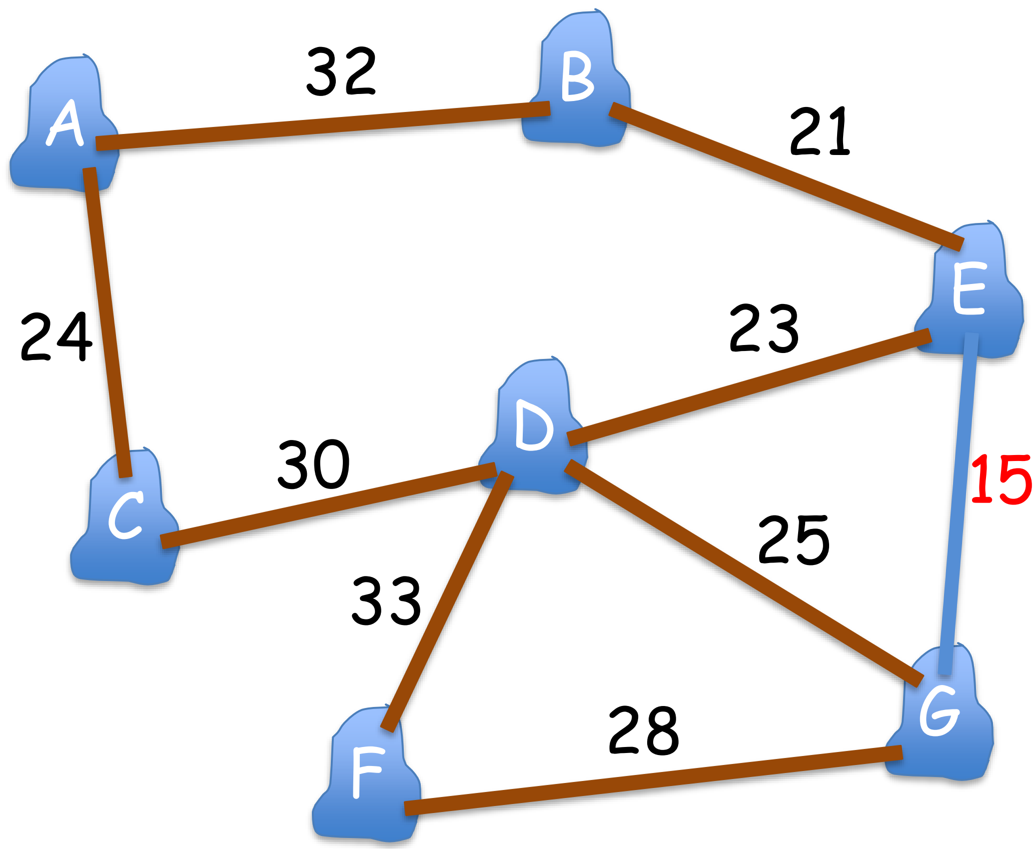
MST



n-1 edges enough to connect n nodes, one more creates a cycle

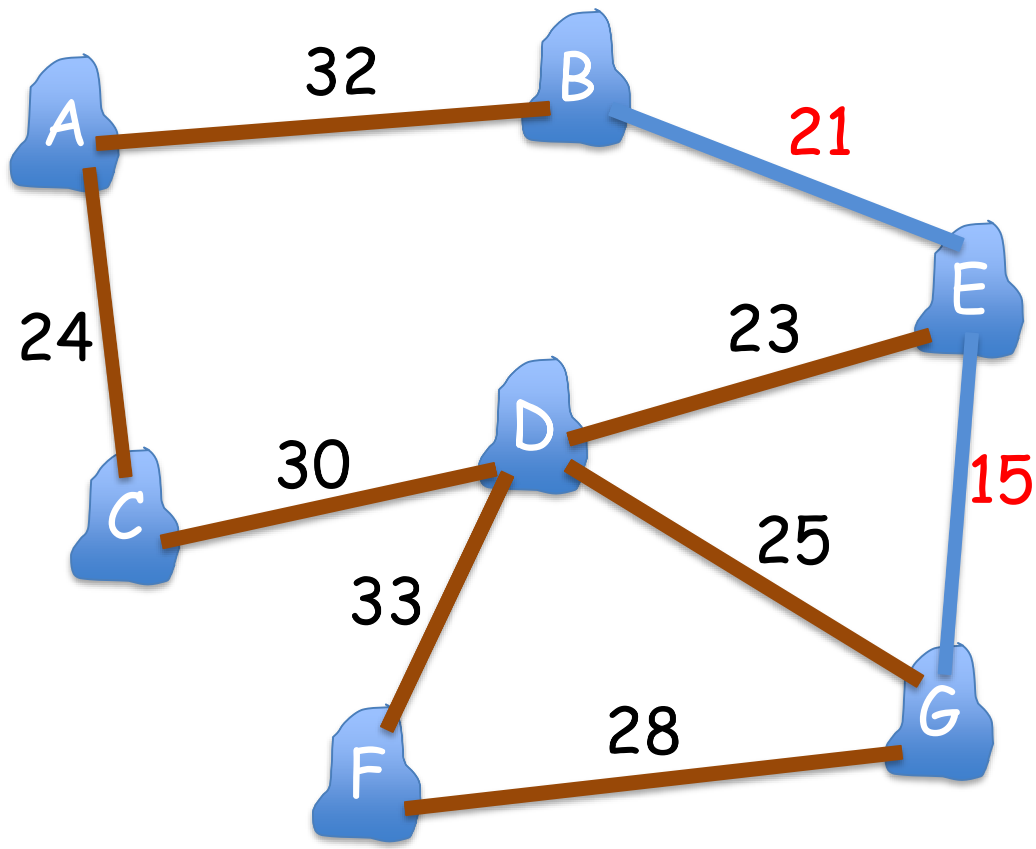
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



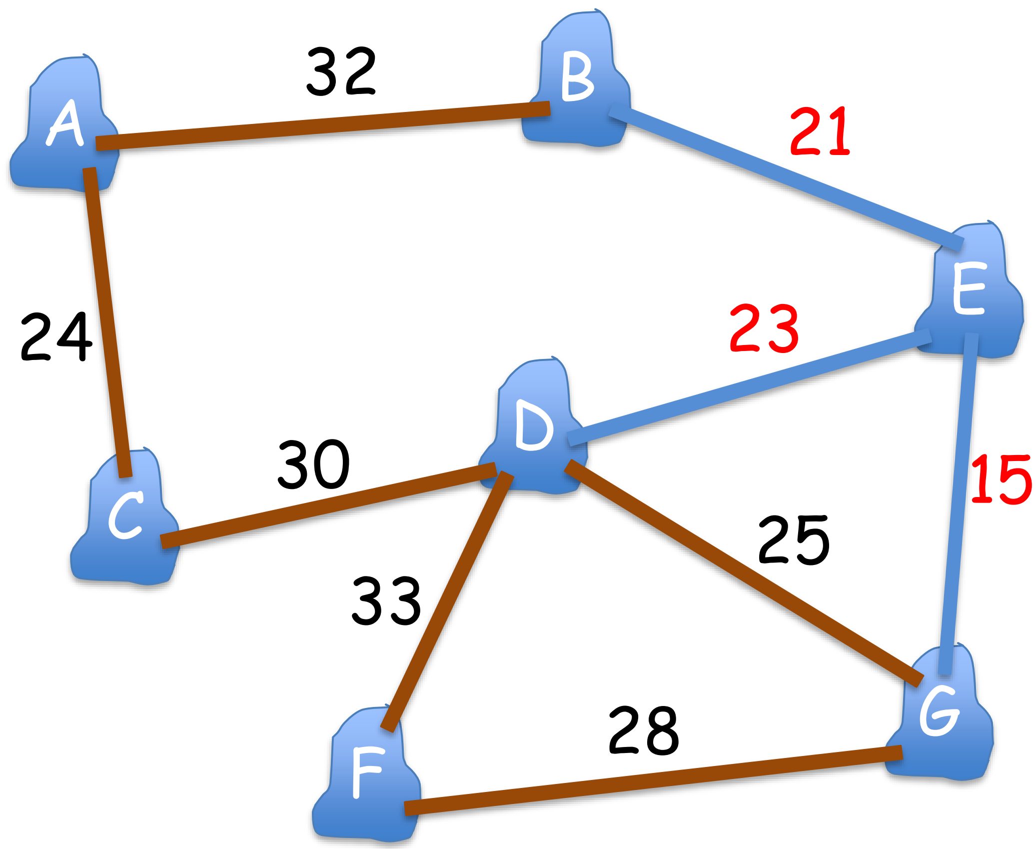
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



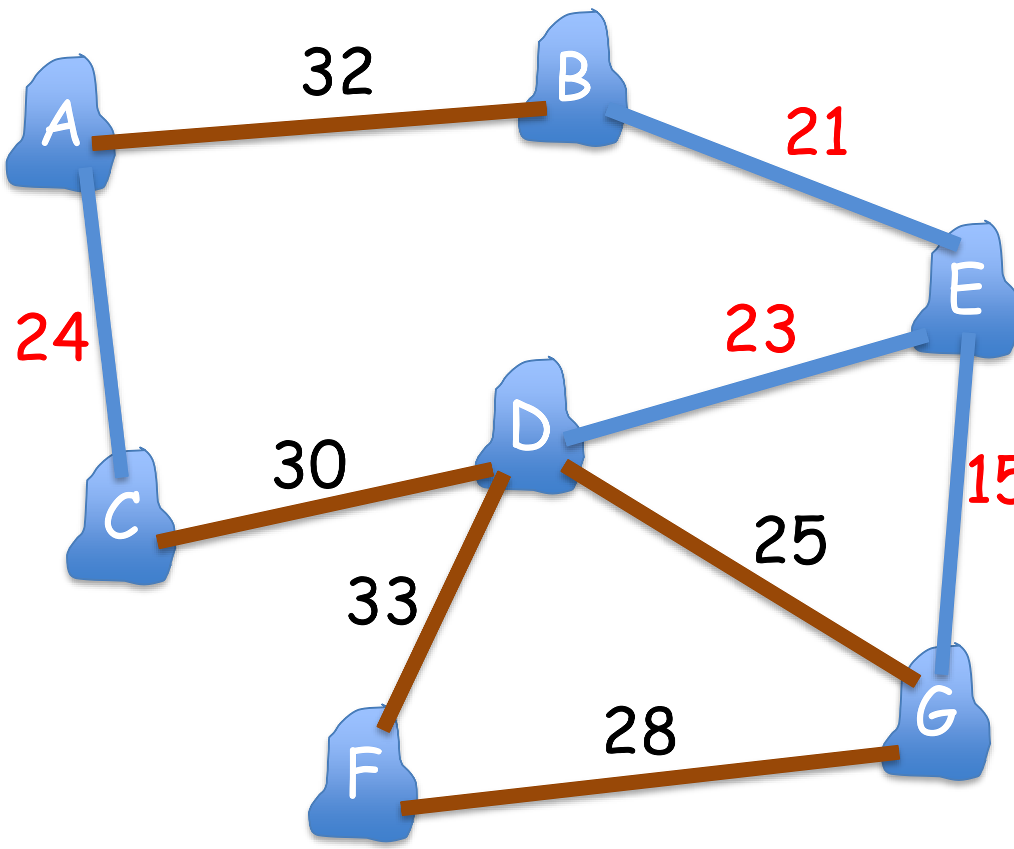
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



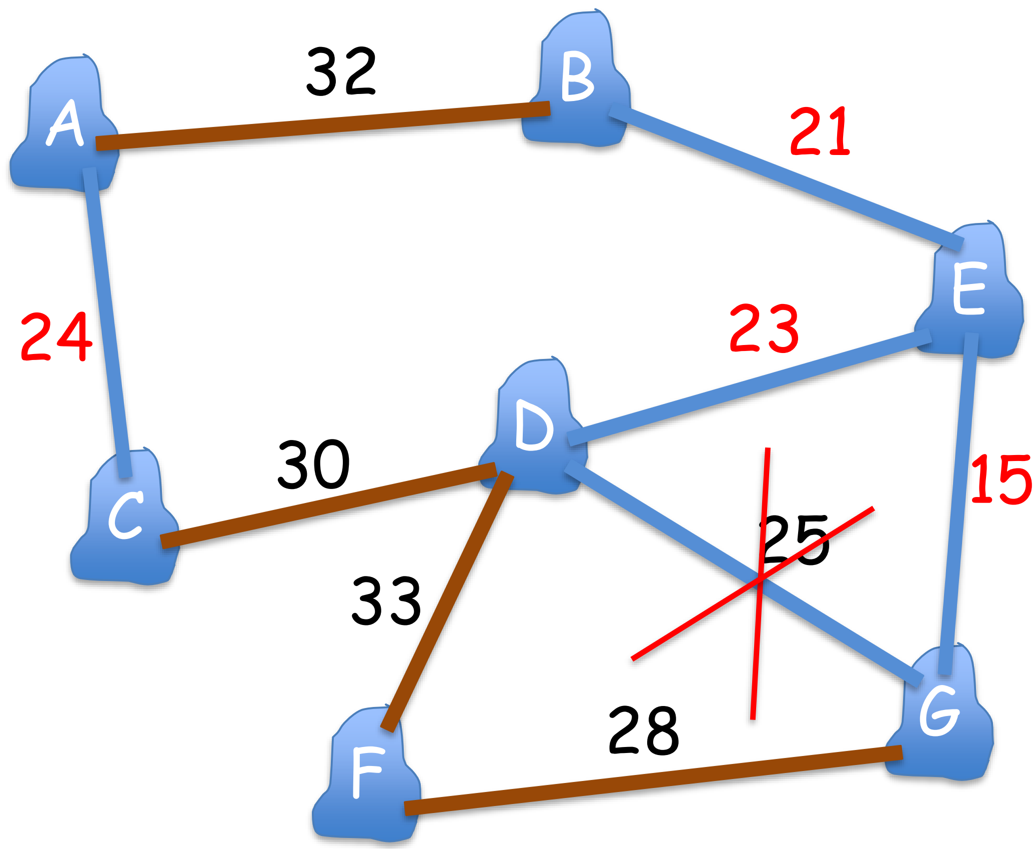
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



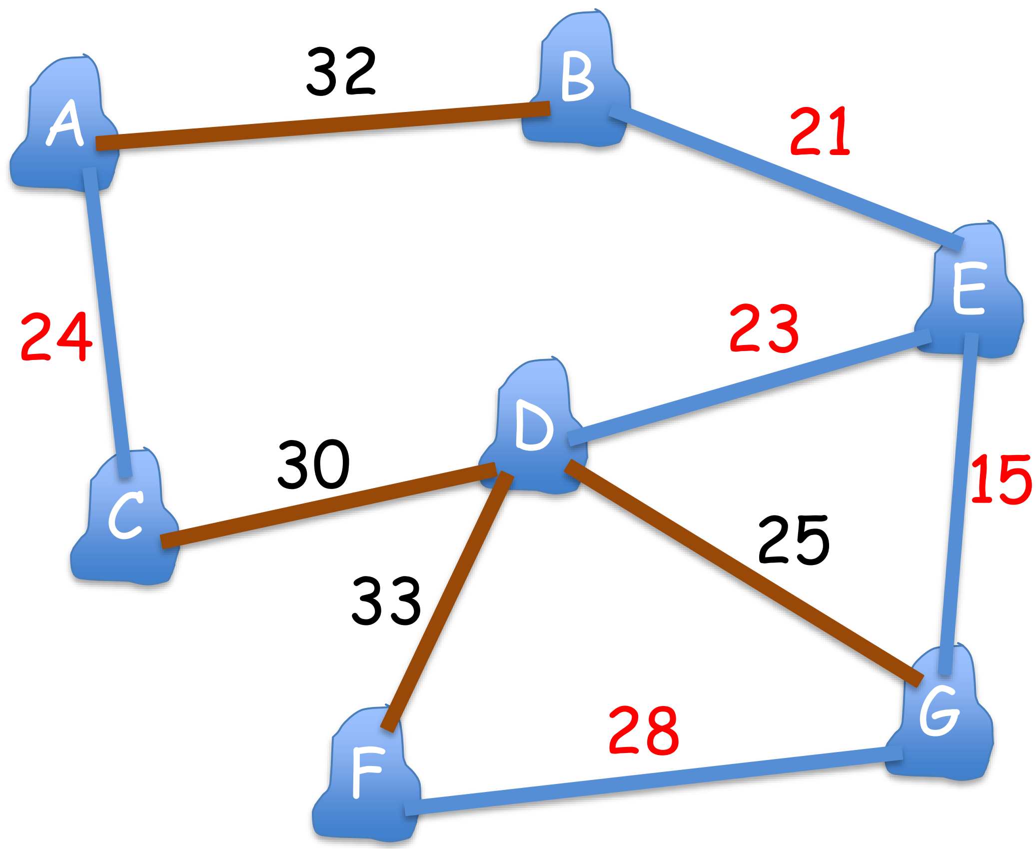
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



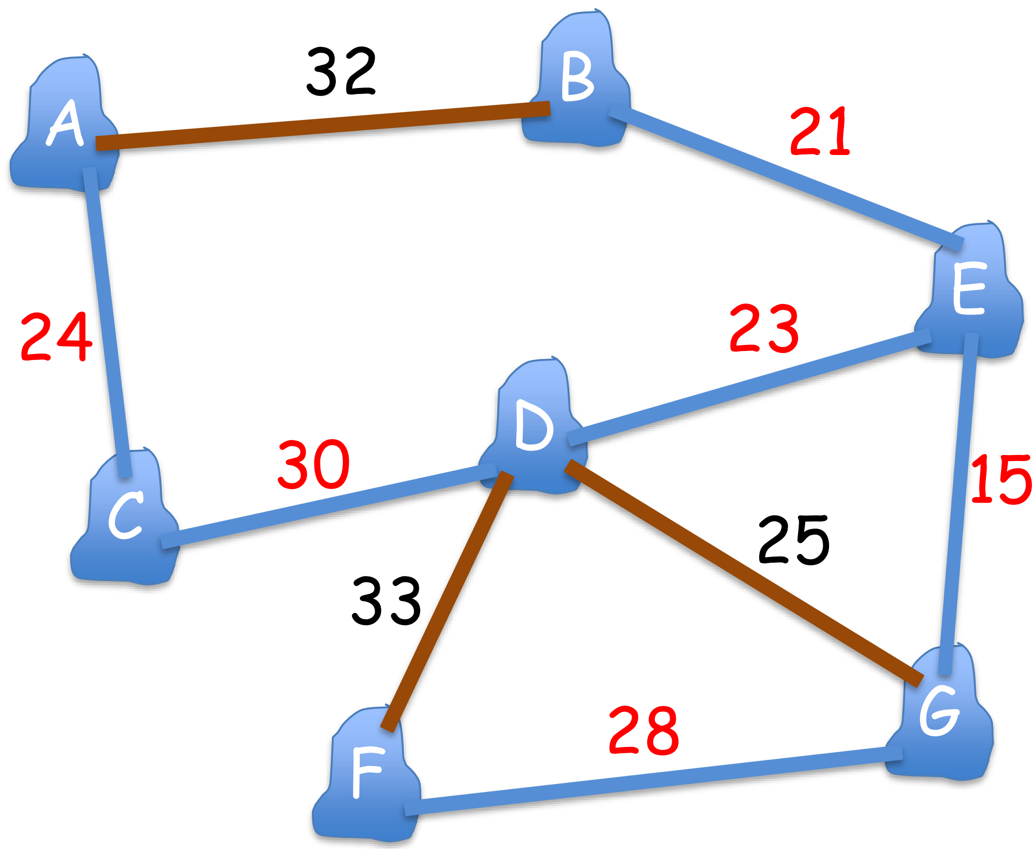
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



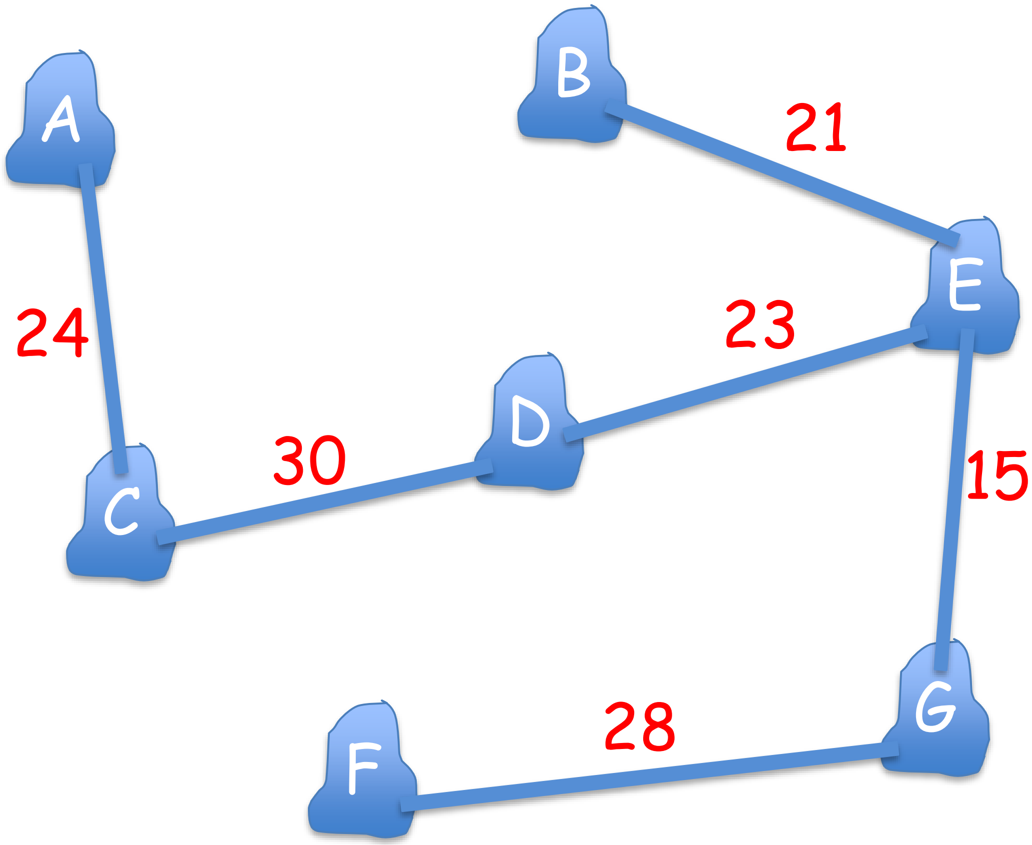
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



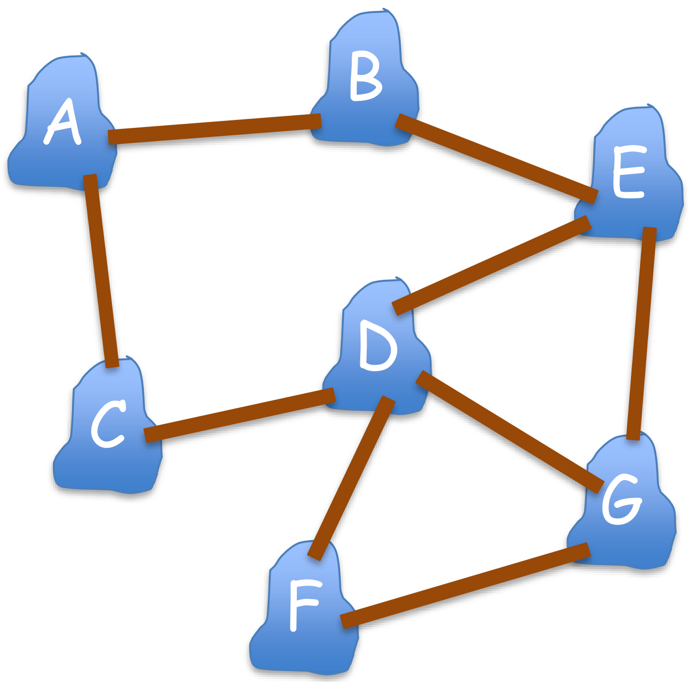
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



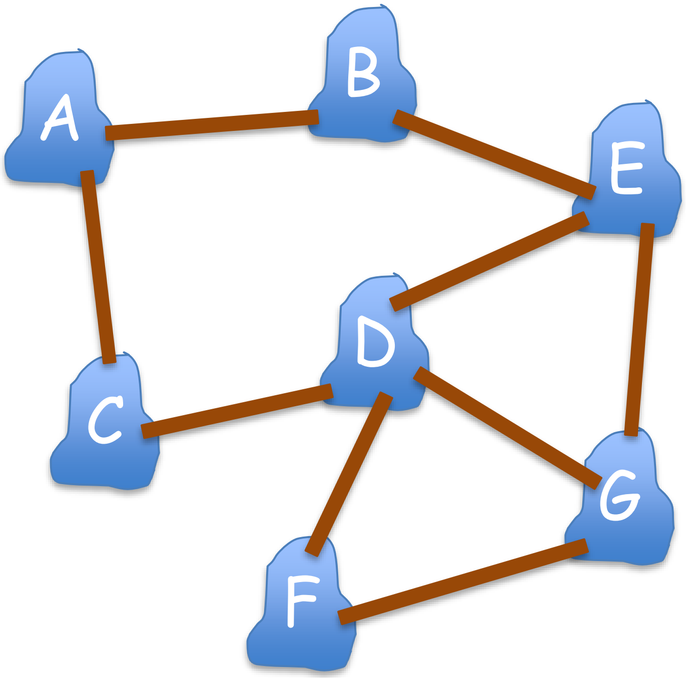
- start with an empty set of edges A
- repeat the following procedure |V| – 1 times :
  add the lightest edge that does not create a cycle to A

MST



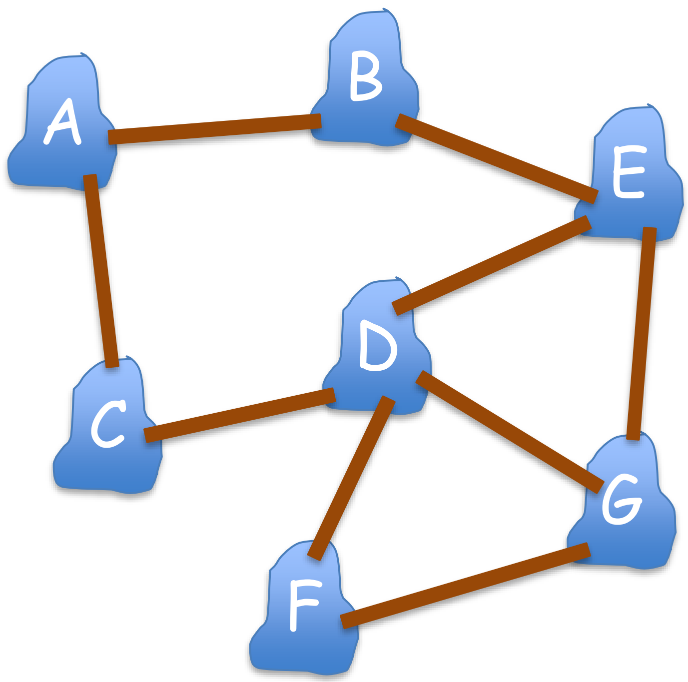
Why does this algorithm work?

MST



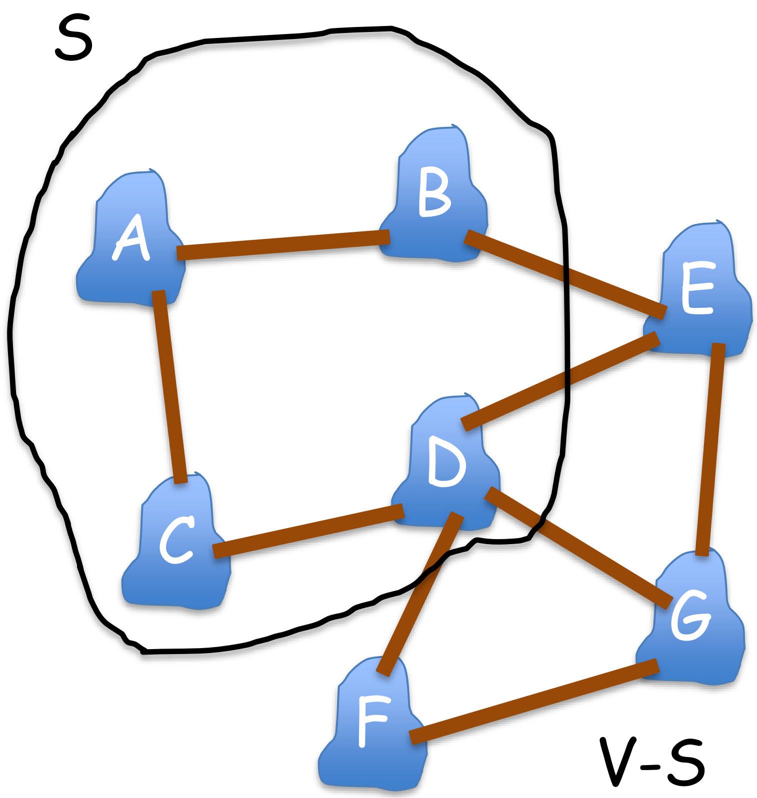
Why does this algorithm work?
(Greedy Choice Property)

MST



Why does this algorithm work?
(Greedy Choice Property)

cut : a cut of G=(V,E) is a partition of V into two sets (S, V-S)

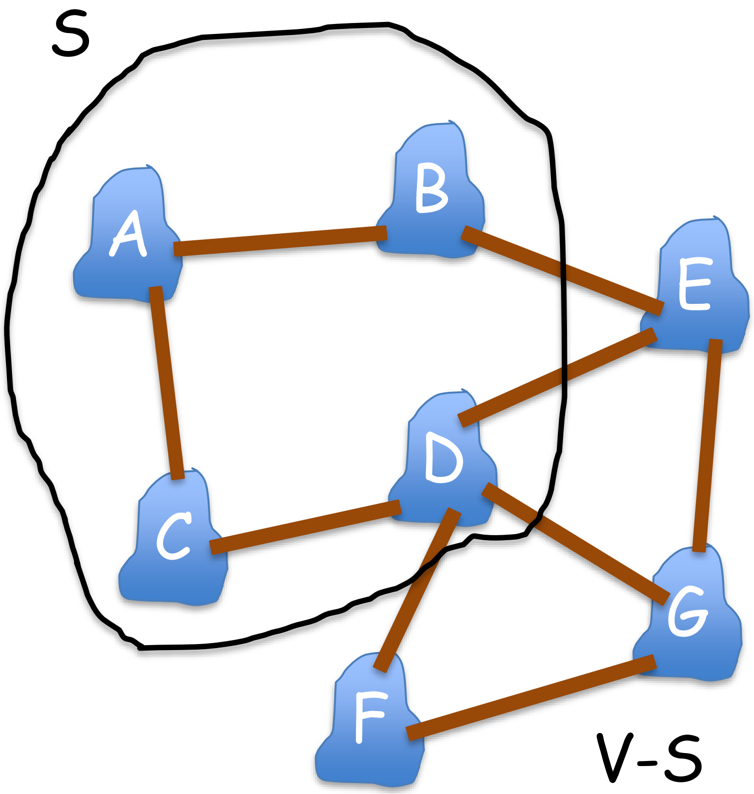


MST

Why does this algorithm work?
(Greedy Choice Property)

cut : a cut of G=(V,E) is a partition of V
into two sets (S, V-S)
{A,B,C,D} and {E,F,G}

MST

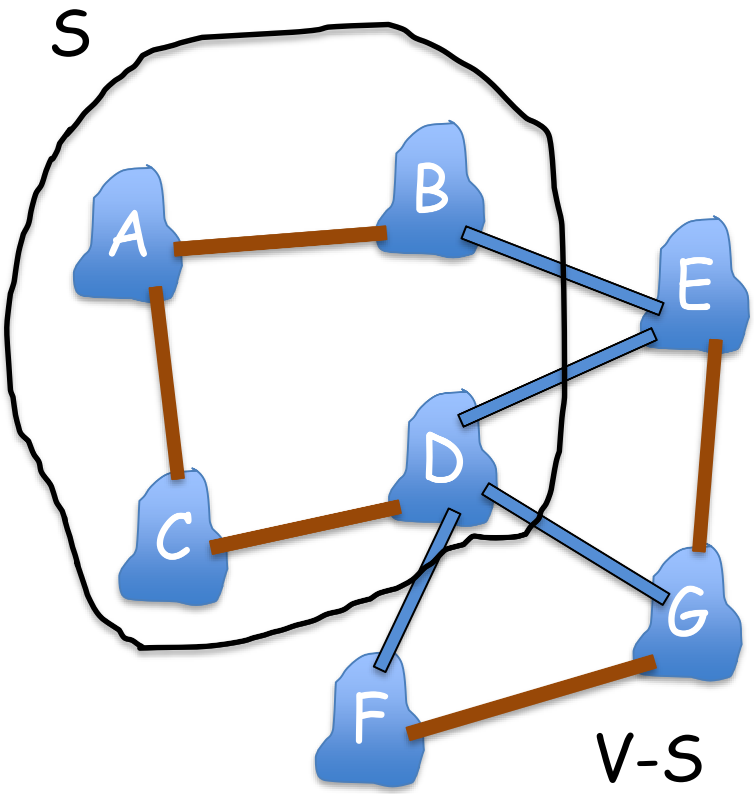


Why does this algorithm work?
(Greedy Choice Property)

cut : a cut of G=(V,E) is a partition of V
into two sets (S, V-S)
{A,B,C,D} and {E,F,G}

crossing a cut : an edge (u,v) crosses a cut
(S,V-S) if u in S and v in V-S

MST

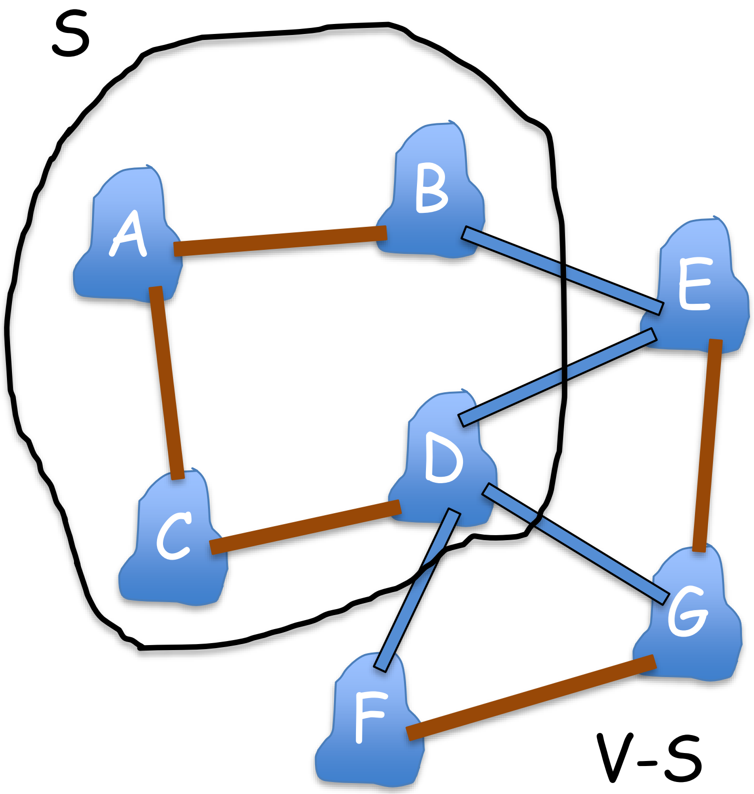


Why does this algorithm work?
(Greedy Choice Property)

cut : a cut of G=(V,E) is a partition of V
into two sets (S, V-S)
{A,B,C,D} and {E,F,G}

crossing a cut : an edge (u,v) crosses a cut
(S,V-S) if u in S and v in V-S
(B,E), (D,E), (D,G), (D,F) crossing a cut

MST

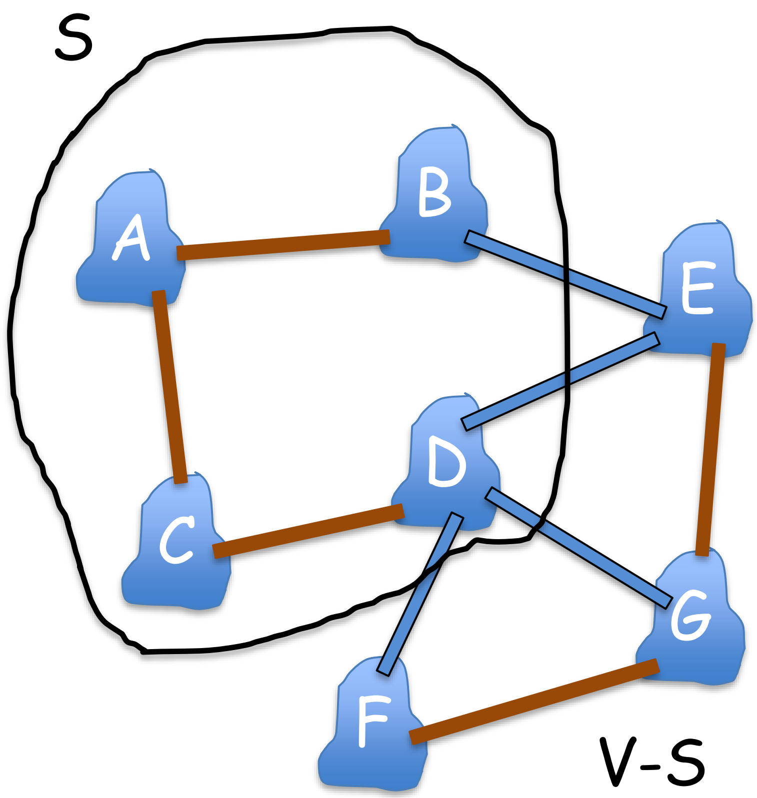


Why does this algorithm work?
(Greedy Choice Property)

cut : a cut of G=(V,E) is a partition of V
into two sets (S, V-S)
{A,B,C,D} and {E,F,G}

crossing a cut : an edge (u,v) crosses a cut
(S,V-S) if u in S and v in V-S
(B,E), (D,E), (D,G), (D,F) crossing a cut

respect : a set A respects a cut (S,V-S) if
no edge in A crosses the cut



MST

Why does this algorithm work?
(Greedy Choice Property)

cut : a cut of G=(V,E) is a partition of V
into two sets (S, V-S)
{A,B,C,D} and {E,F,G}

crossing a cut : an edge (u,v) crosses a cut
(S,V-S) if u in S and v in V-S
(B,E), (D,E), (D,G), (D,F) crossing a cut

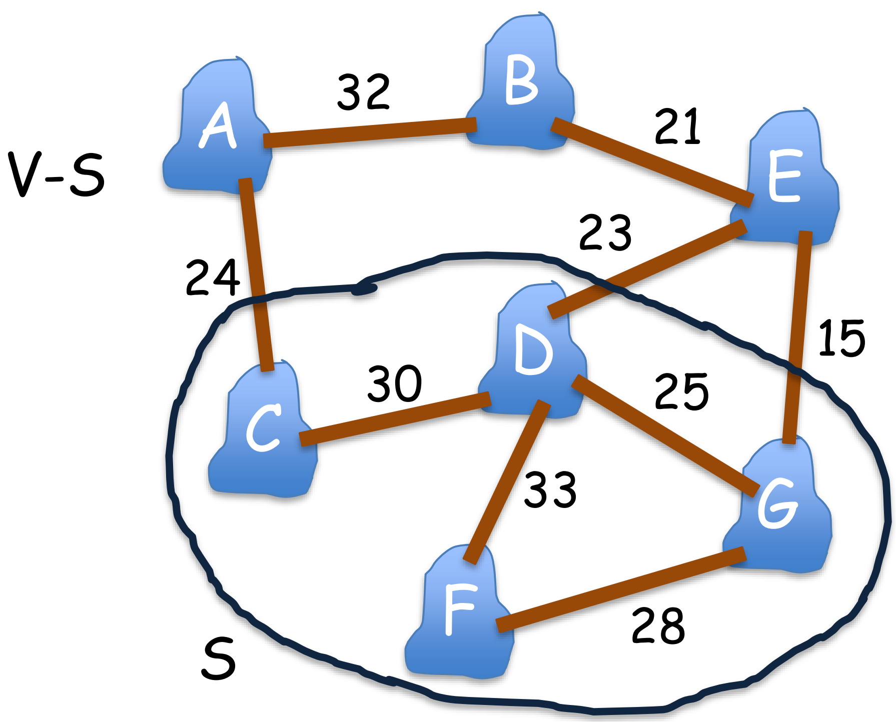
respect : a set A respects a cut (S,V-S) if
no edge in A crosses the cut
A={(E,G),(F,G)} respects (S,V-S)

Greedy Choice Property

<u>Theorem</u> : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

Greedy Choice Property

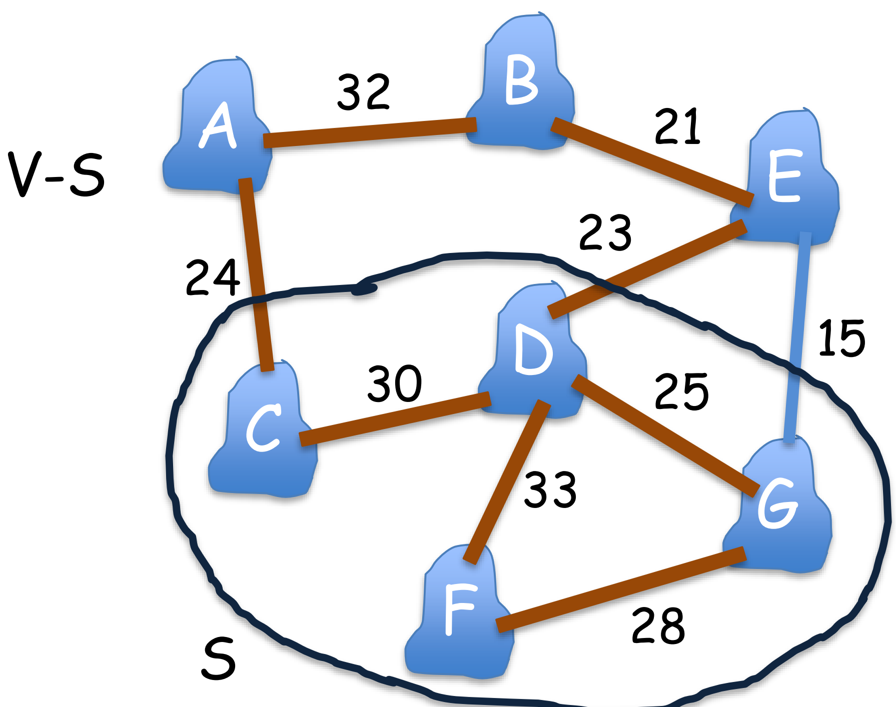
<u>Theorem</u> : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.



A = {(F,G),(B,E)}

Greedy Choice Property

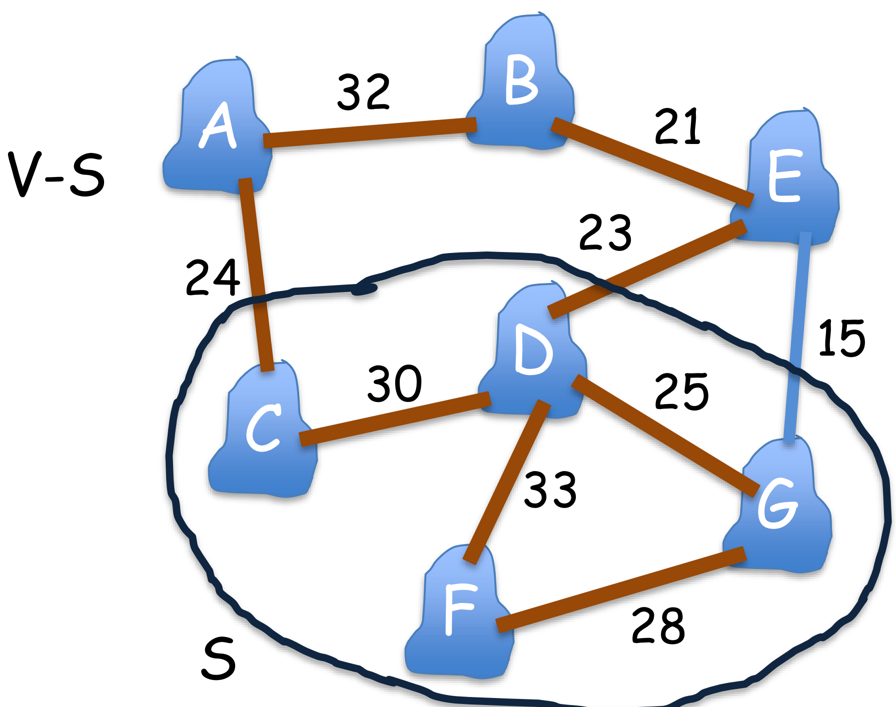
Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.



A = {(F,G),(B,E)}

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

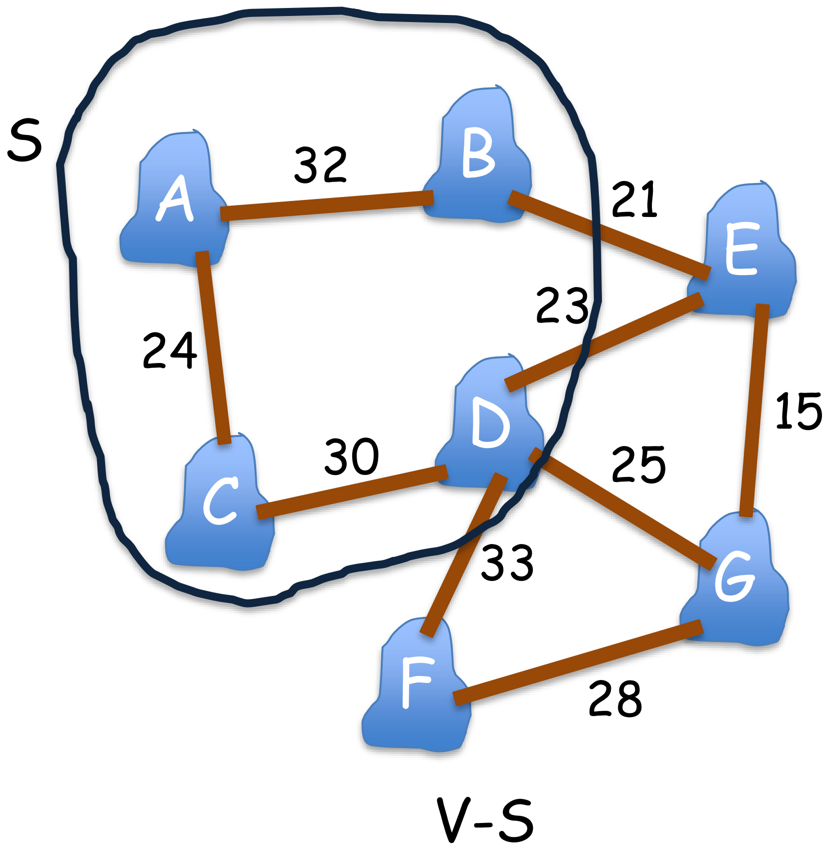


A = {(F,G),(B,E)}

A ∪ {(E,G)}

Greedy Choice Property

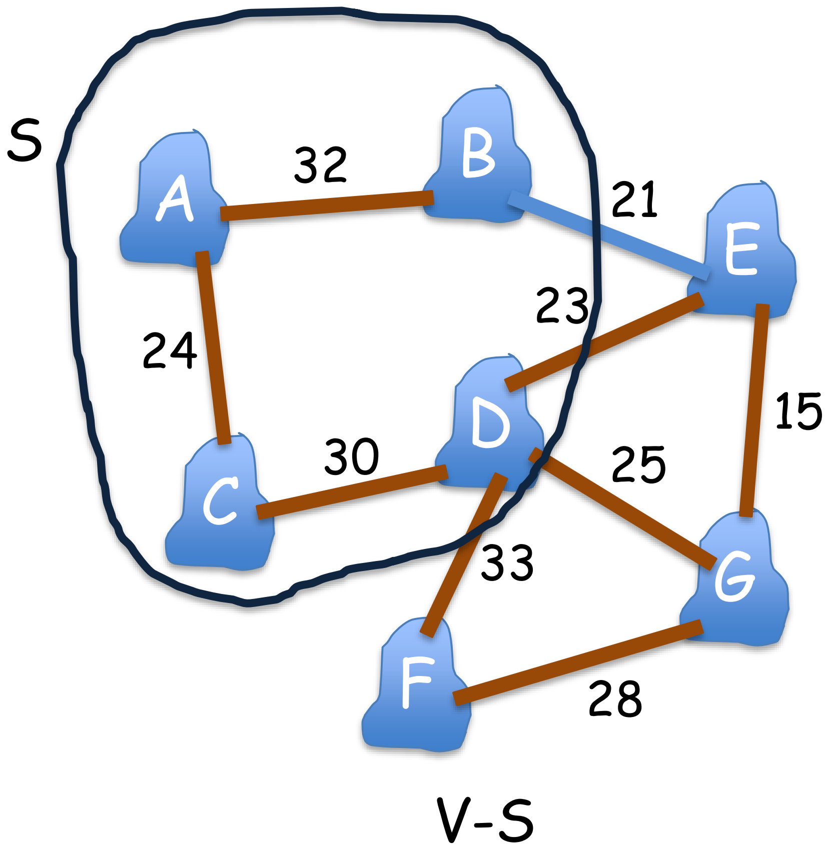
Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.



A = {(A,C),(E,G)}

Greedy Choice Property

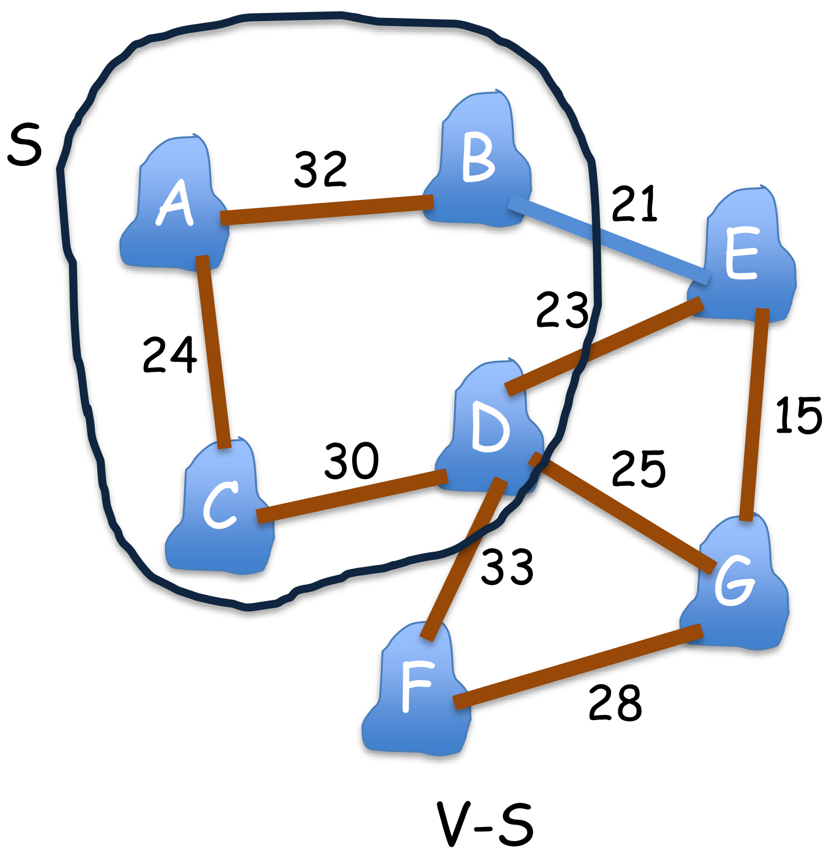
Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.



A = {(A,C),(E,G)}

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

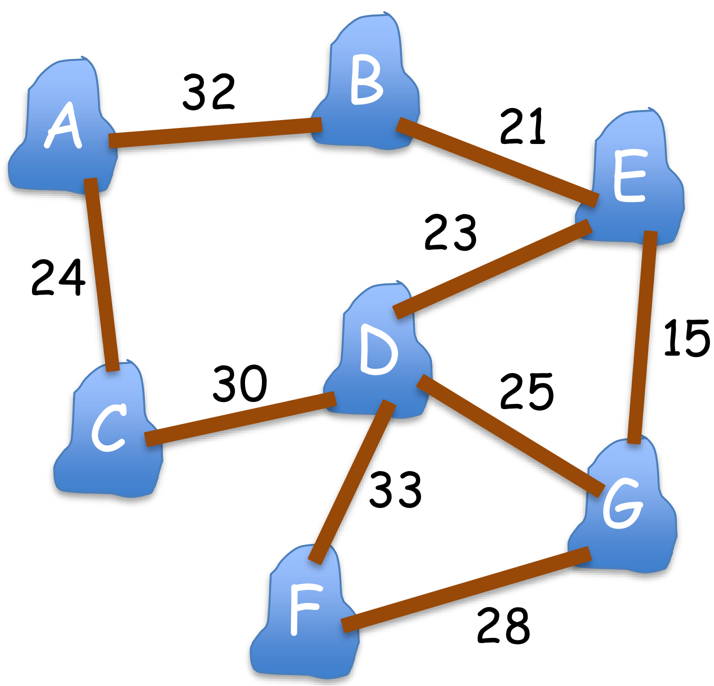


A = {(A,C),(E,G)}

A ∪ {(B,E)}

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

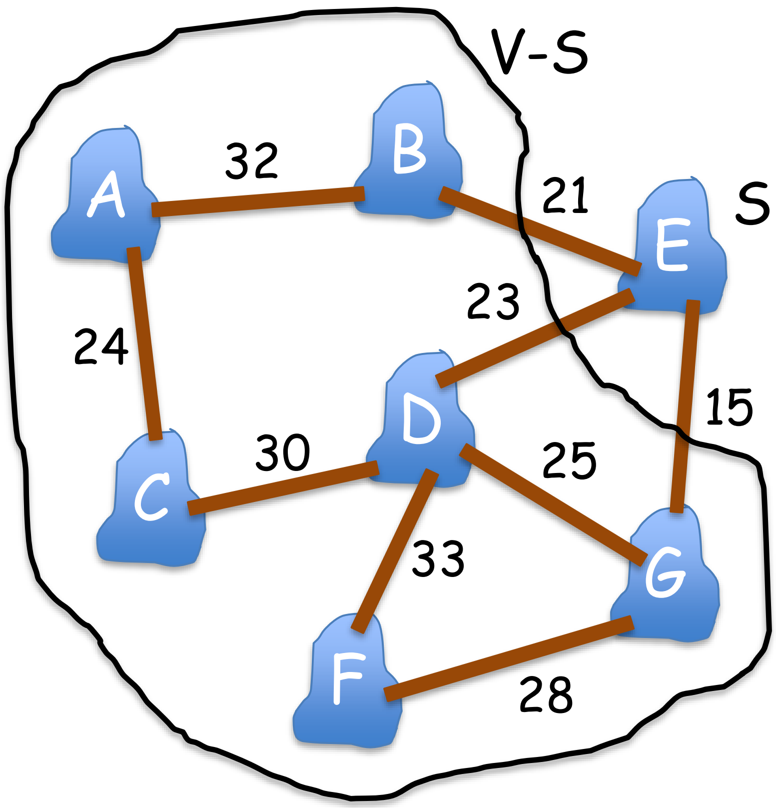


A = { }

- start with any node

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

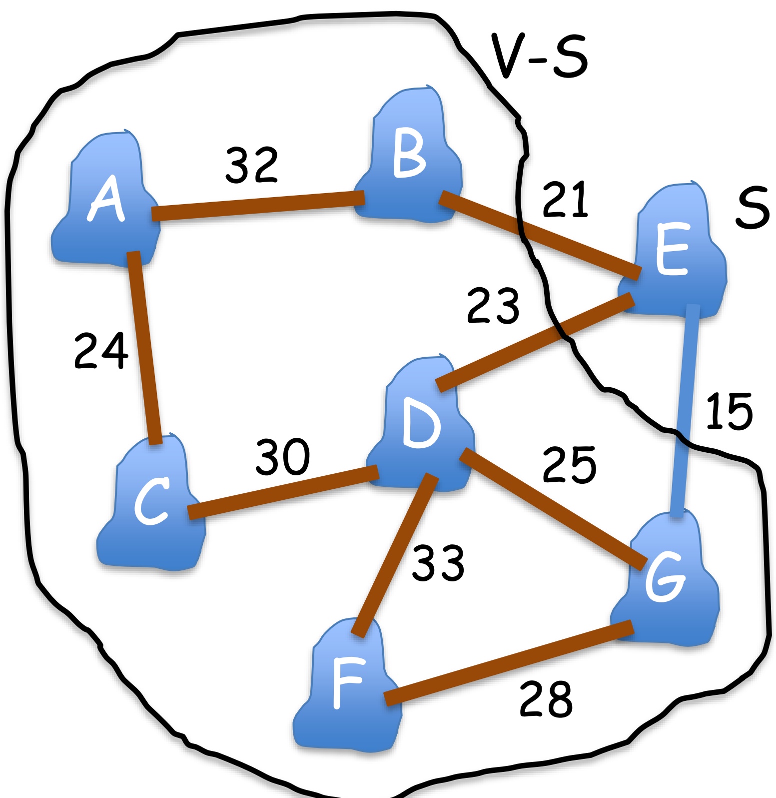


A = { }

- start with any node
- A respects V-S (or S)

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

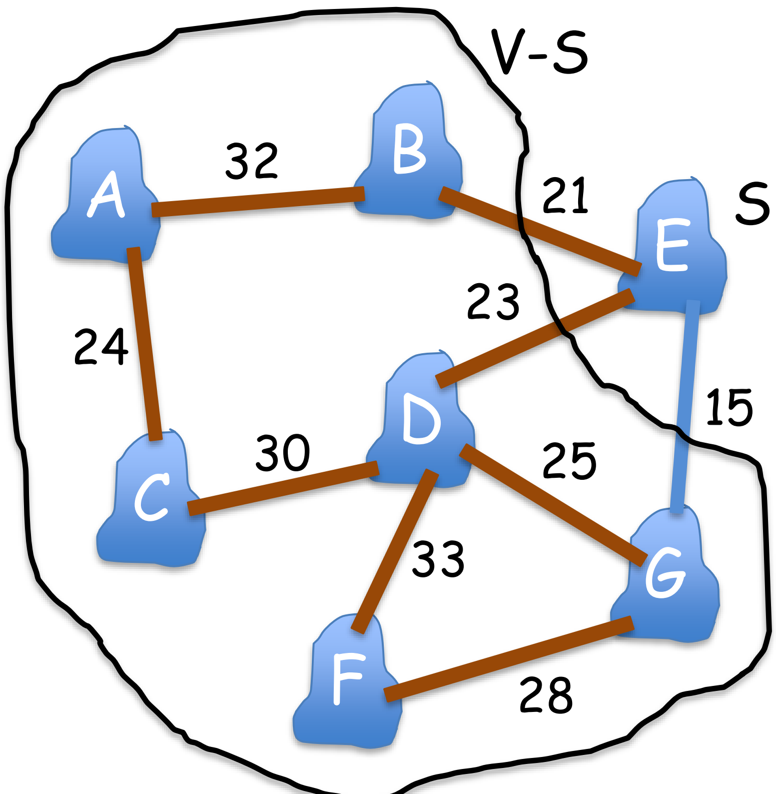


A = { }

- start with any node
- A respects V-S (or S)
- (E,G) is the lightest edge that crosses (S,V-S)

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

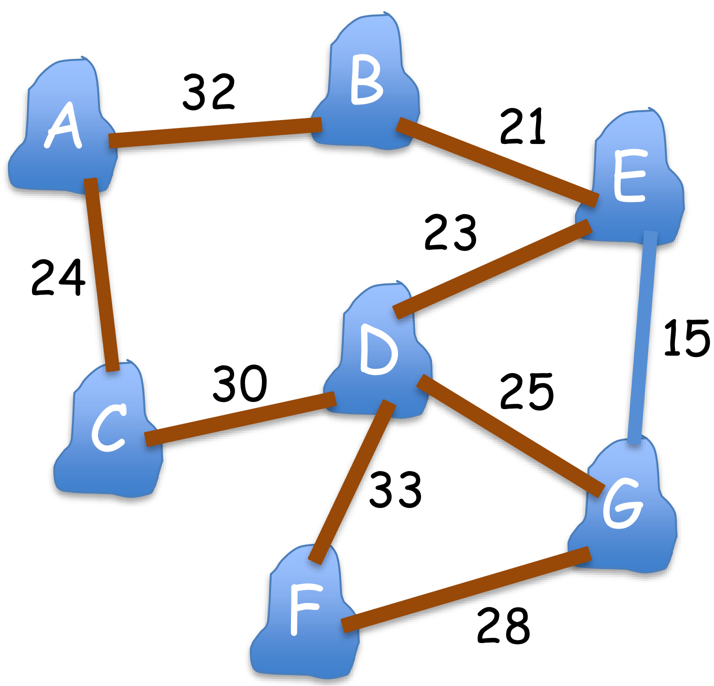


A = { }

- start with any node
- A respects V-S (or S)
- (E,G) is the lightest edge that crosses (S,V-S)
- A = A ∪ {(E,G)}

Greedy Choice Property

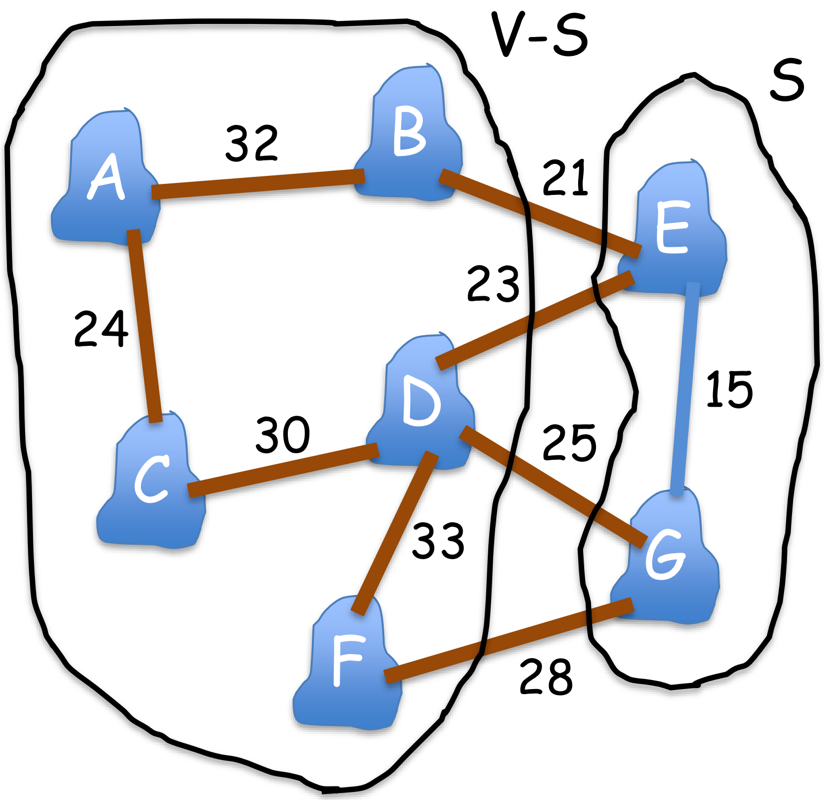
Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.



A = {(E,G)}

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then $A \cup \{e\}$ is a subset of some minimum spanning tree.

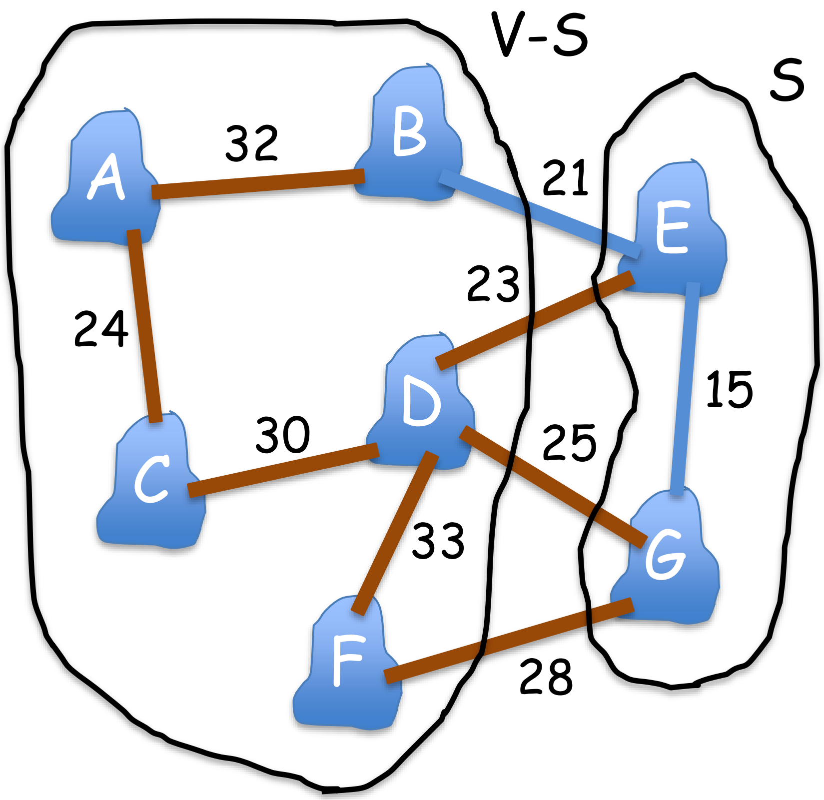


$A = \{(E,G)\}$

- A respects V-S (or S)

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

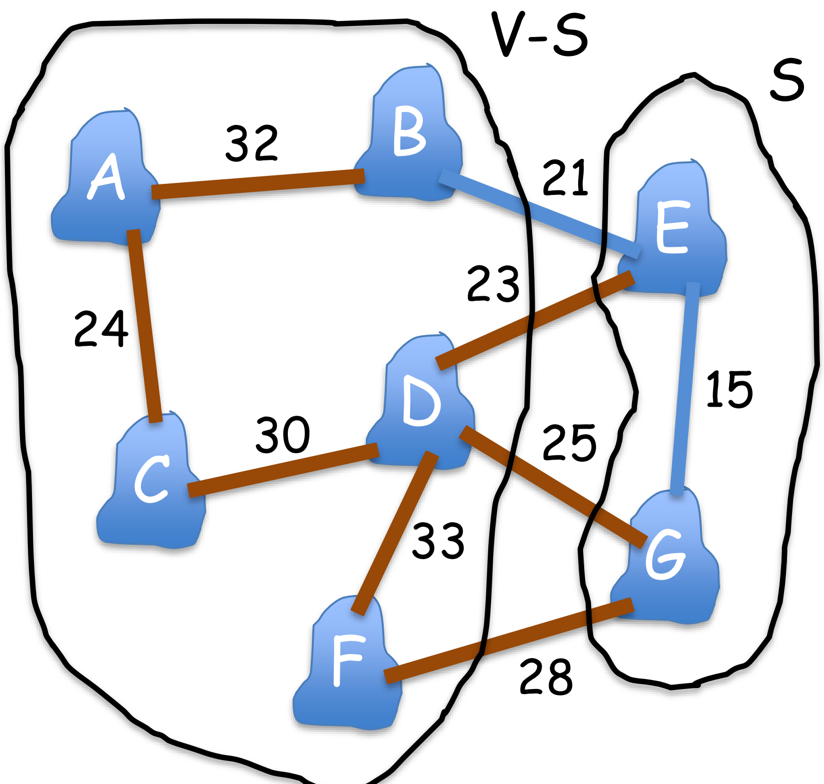


A = {(E,G)}

- A respects V-S (or S)
- (B,E) is the lightest edge that crosses (S,V-S)

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

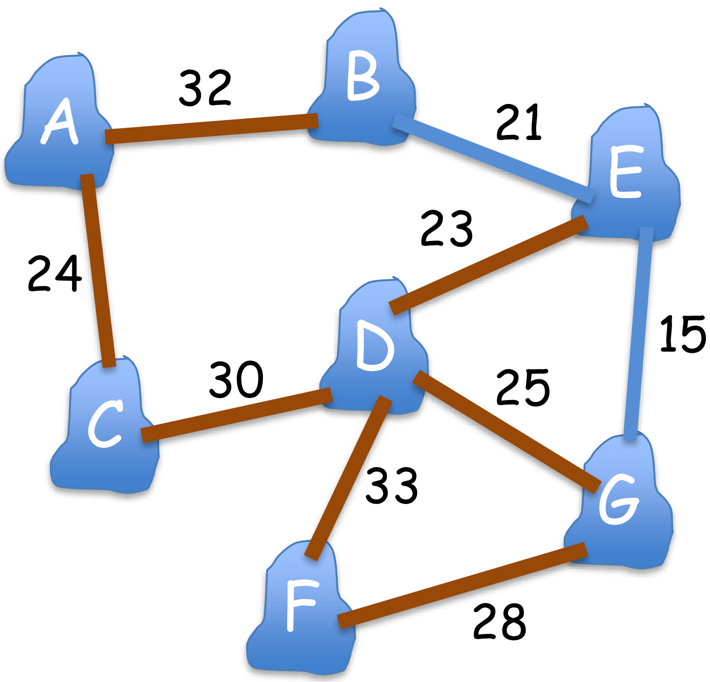


A = {(E,G)}

- A respects V-S (or S)
- (B,E) is the lightest edge that crosses (S,V-S)
- A = A ∪ {(B,E)}

Greedy Choice Property

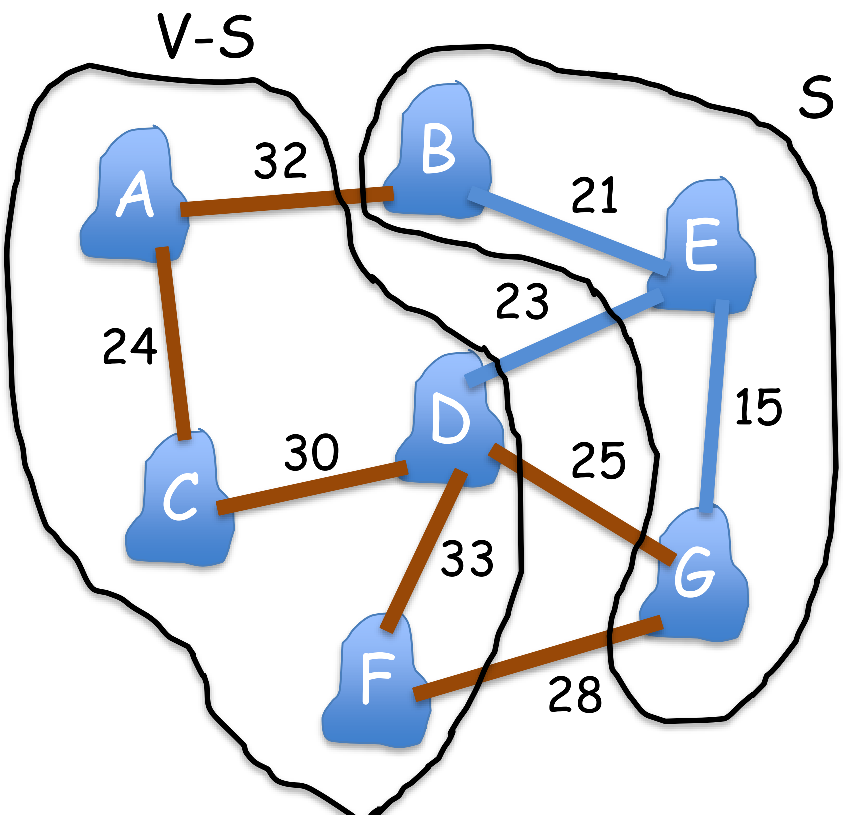
<u>Theorem</u> : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.



A = {(E,G),(B,E)}

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

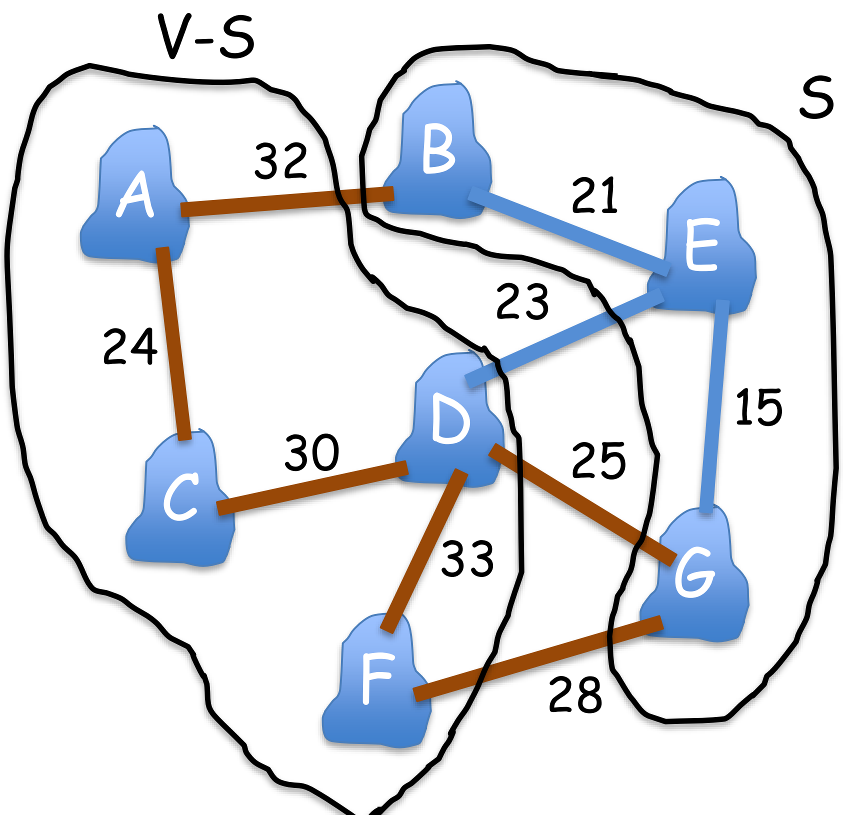


A = {(E,G),(B,E)}

- A respects V-S (or S)
- (D,E) is the lightest edge that crosses (S,V-S)

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.



A = {(E,G),(B,E)}

- A respects V-S (or S)
- (D,E) is the lightest edge that crosses (S,V-S)
- A = A ∪ {(D,E)}

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let $(S,V-S)$ be a cut that A respects and e be the lightest edge that crosses $(S,V-S)$. Then $A\cup\{e\}$ is a subset of some minimum spanning tree.

Proof : Assume T is an MST and $\{e\}$ is contained in T.

Greedy Choice Property

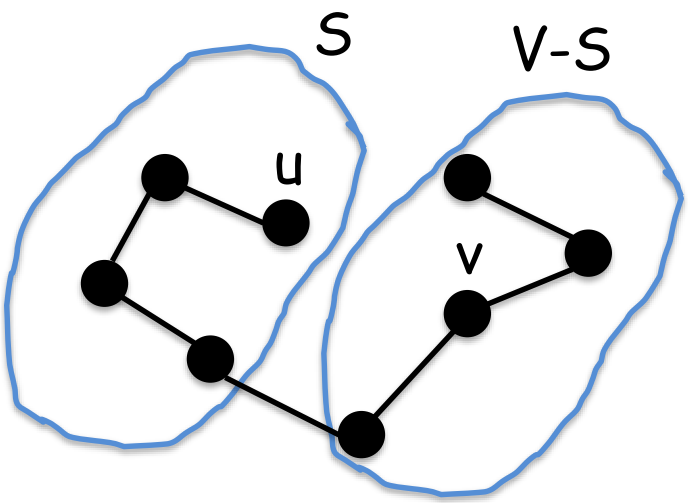
Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then $A \cup \{e\}$ is a subset of some minimum spanning tree.

Proof : Assume T is an MST and {e} is contained in T. Let e=(u,v) . Since T is an MST, T must contain a path from v to u.

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

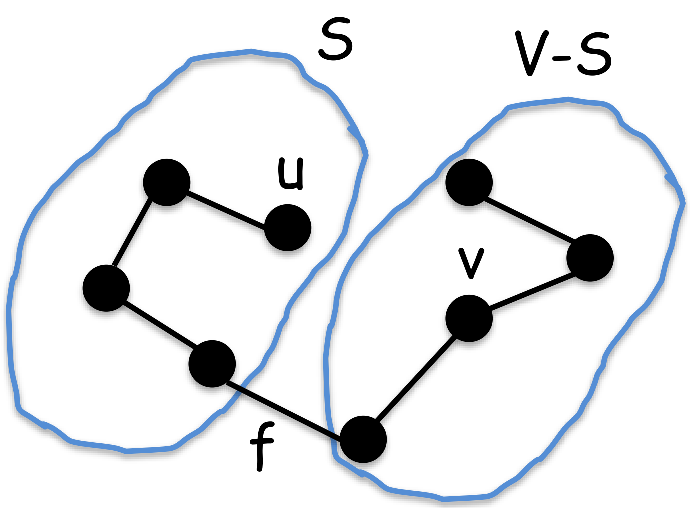
Proof : Assume T is an MST and {e} is contained in T. Let e=(u,v) . Since T is an MST, T must contain a path from v to u.



Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then $A \cup \{e\}$ is a subset of some minimum spanning tree.

Proof : Assume T is an MST and {e} is contained in T. Let $e=(u,v)$. Since T is an MST, T must contain a path from v to u.

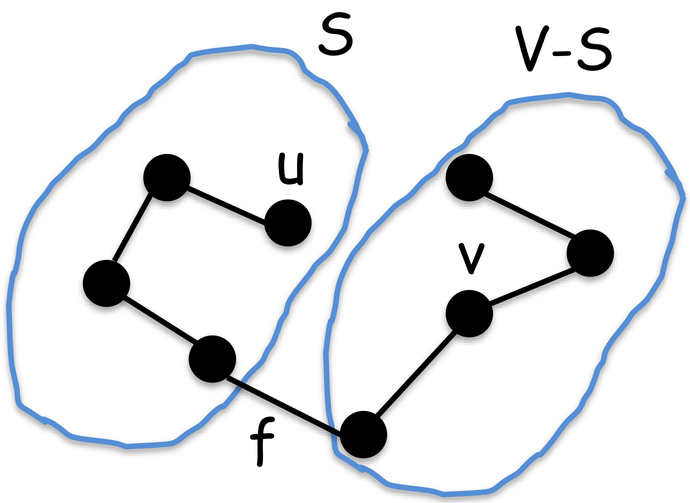


- Let f be the edge on this path that crosses (S,V-S).

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

Proof : *Assume T is an MST and {e} is contained in T. Let e=(u,v) .* Since T is an MST, T must contain a path from v to u.

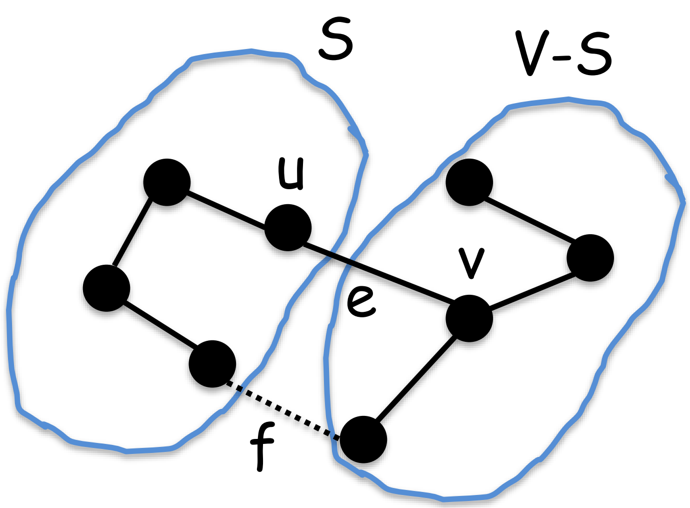


- Let f be the edge on this path that crosses (S,V-S). Such edge must exist since there should be an edge that connects this cut

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

Proof : Assume T is an MST and {e} is contained in T. Let e=(u,v) . Since T is an MST, T must contain a path from v to u.



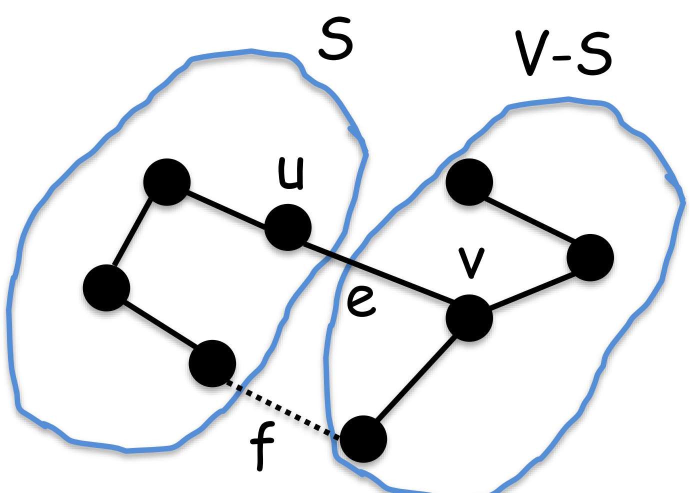
- Let f be the edge on this path that crosses (S,V-S). Such edge must exist since there should be an edge that connects this cut
- define a new MST T' as

$$T' = T \cup \{e\} - \{f\}$$

Greedy Choice Property

<u>Theorem</u> : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

Proof : *Assume T is an MST and {e} is contained in T. Let e=(u,v) .* Since T is an MST, T must contain a path from v to u.



- Let f be the edge on this path that crosses (S,V-S). Such edge must exist since there should be an edge that connects this cut
- define a new MST T' as

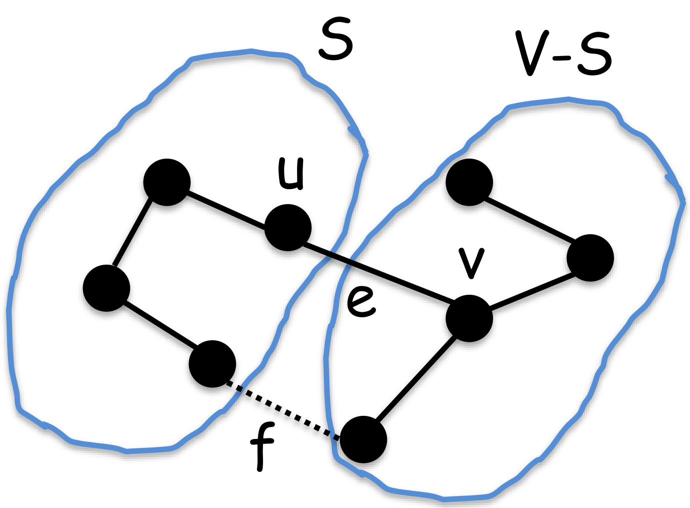
  T' = T ∪{e} – {f}
- *since e is the lightest edge that crosses (S,V-S) (from assumption),*

  w(e) ≤ w(f)

Greedy Choice Property

<u>Theorem</u> : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then A∪{e} is a subset of some minimum spanning tree.

Proof : Assume T is an MST and {e} is contained in T. Let e=(u,v) . Since T is an MST, T must contain a path from v to u.



- Let f be the edge on this path that crosses (S,V-S). Such edge must exist since there should be an edge that connects this cut
- define a new MST T ' as

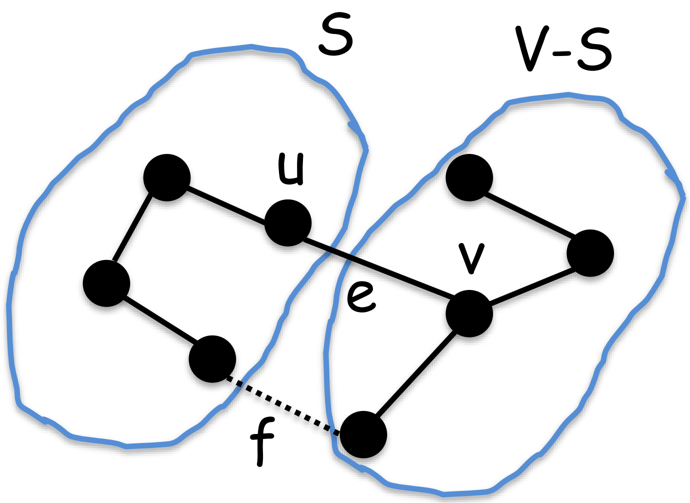
  $$T' = T \cup \{e\} - \{f\}$$
- since e is the lightest edge that crosses (S,V-S) (from assumption),

  $$w(e) \leq w(f)$$
- Thus, $w(T') = w(T) - w(f) + w(e) \leq w(T)$

Greedy Choice Property

Theorem : Let A be a subset of edges of some minimum spanning tree T. Let (S,V-S) be a cut that A respects and e be the lightest edge that crosses (S,V-S). Then $A \cup \{e\}$ is a subset of some minimum spanning tree.

Proof : Assume T is an MST and {e} is contained in T. Let e=(u,v) . Since T is an MST, T must contain a path from v to u.



- Let f be the edge on this path that crosses (S,V-S). Such edge must exist since there should be an edge that connects this cut
- define a new MST T ' as
  $$T' = T \cup \{e\} - \{f\}$$
- since e is the lightest edge that crosses (S,V-S) (from assumption),
  $$w(e) \leq w(f)$$
- Thus, $w(T') = w(T) - w(f) + w(e) \leq w(T)$

this is a contradiction !

Kruskal's Algorithm

- start with an empty tree T
- consider edges in increasing order of weights
- add the lightest edge to T if it does not create a cycle
- If It creates a cycle, discard it

Kruskal's Algorithm

- start with an empty tree T
- consider edges in increasing order of weights
- add the lightest edge to T if it does not create a cycle
- If It creates a cycle, discard it

How to check whether adding an edge to T will create a cycle?

Kruskal's Algorithm

- start with an empty tree T
- consider edges in increasing order of weights
- add the lightest edge to T if it does not create a cycle
- If It creates a cycle, discard it

How to check whether adding an edge to T will create a cycle?

- DFS can be used, but it takes $O(|E| + |V|)$ time for each step, $O(|E|.|V|)$ at total

Kruskal's Algorithm

- start with an empty tree T
- consider edges in increasing order of weights
- add the lightest edge to T if it does not create a cycle
- If It creates a cycle, discard it

How to check whether adding an edge to T will create a cycle?

- DFS can be used, but it takes $O(|E| + |V|)$ time for each step, $O(|E|.|V|)$ at total

- use union-find data structure

Union-Find

- represent each set as a tree
- each node has a pointer to its parent
- the root has a parent pointer to itself
- each tree has a height

Union-Find

- represent each set as a tree
- each node has a pointer to its parent
- the root has a parent pointer to itself
- each tree has a height
- data structure supports three operations

Union-Find

- represent each set as a tree
- each node has a pointer to its parent
- the root has a parent pointer to itself
- each tree has a height
- data structure supports three operations

Make-Set(x)
x.parent = x
x.height = 0

Union-Find

- represent each set as a tree
- each node has a pointer to its parent
- the root has a parent pointer to itself
- each tree has a height
- data structure supports three operations

Make-Set(x)
```
x.parent = x
x.height = 0
```

Find-Set(x)
```
while x ≠ x.parent
      x = x.parent
return x
```

Union-Find

- represent each set as a tree
- each node has a pointer to its parent
- the root has a parent pointer to itself
- each tree has a height
- data structure supports three operations

Make-Set(x)
x.parent = x
x.height = 0

Find-Set(x)
```
while x ≠ x.parent
      x = x.parent
return x
```

return the root of the set

Union-Find

- represent each set as a tree
- each node has a pointer to its parent
- the root has a parent pointer to itself
- each tree has a height
- data structure supports three operations

<u>Make-Set(x)</u>

```
x.parent = x
x.height = 0
```

<u>Find-Set(x)</u>

```
while x ≠ x.parent
    x = x.parent
return x
```

return the root of the set

<u>Union(x,y)</u>

```
a = Find-Set(x)
b = Find-Set(y)
if (a.height ≤ b.height)
    if (a.height = b.height)
        b.height = b.height + 1
    a.parent = b
else
    b.parent = a
```

Union-Find

- represent each set as a tree
- each node has a pointer to its parent
- the root has a parent pointer to itself
- each tree has a height
- data structure supports three operations

Make-Set(x)

```
x.parent = x
x.height = 0
```

Find-Set(x)

```
while x ≠ x.parent
      x = x.parent
return x
```

return the root of the set

Union(x,y)

```
a = Find-Set(x)
b = Find-Set(y)
if (a.height ≤ b.height)
     if (a.height = b.height)
          b.height = b.height + 1
     a.parent = b
else
     b.parent = a
```

Always make the root of the taller one the parent of the shorter

Kruskal's Algorithm

MST-Kruskal(G)

```
A = { }
for each v of V
     Make-Set(v)
sort all edges in increasing
order by weight
for each edge (u,v)
      if Find-Set(u) ≠ Find-Set(v)
           A = A ∪ {u,v}
           Union(u,v)
return A
```

Kruskal's Algorithm

MST-Kruskal(G)

```
A = { }
for each v of V                            O(|V|)
     Make-Set(v)
sort all edges in increasing
order by weight
for each edge (u,v)
      if Find-Set(u) ≠ Find-Set(v)
           A = A ∪ {u,v}
           Union(u,v)
return A
```

Kruskal's Algorithm

MST-Kruskal(G)

```
A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing
order by weight
for each edge (u,v)
     if Find-Set(u) ≠ Find-Set(v)
          A = A ∪ {u,v}
          Union(u,v)
return A
```

O(|V|)

O(|E|.log|E|)

Kruskal's Algorithm

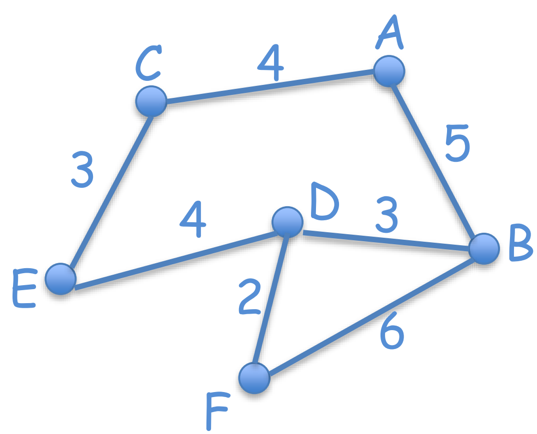
<u>MST-Kruskal(G)</u>

```
A = { }
for each v of V                           O(|V|)
    Make-Set(v)
sort all edges in increasing              O(|E|.log|E|)
order by weight
for each edge (u,v)
     if Find-Set(u) ≠ Find-Set(v)
          A = A ∪ {u,v}                   O(|E|.log|V|)
          Union(u,v)
return A
```

Kruskal's Algorithm

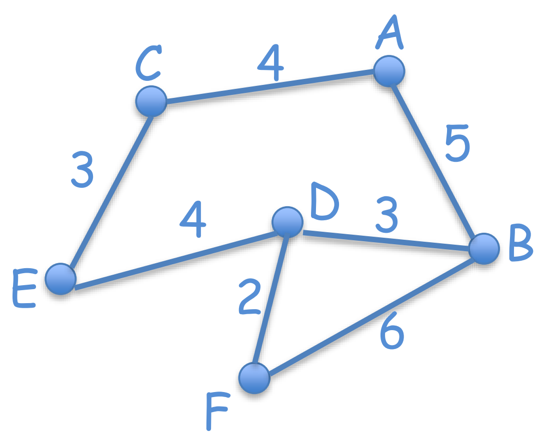
<u>MST-Kruskal(G)</u>

```
A = { }
for each v of V
     Make-Set(v)
sort all edges in increasing order by
weight
for each edge (u,v)
      if Find-Set(u) ≠ Find-Set(v)
           A = A ∪ {u,v}
           Union(u,v)
return A
```



Kruskal's Algorithm

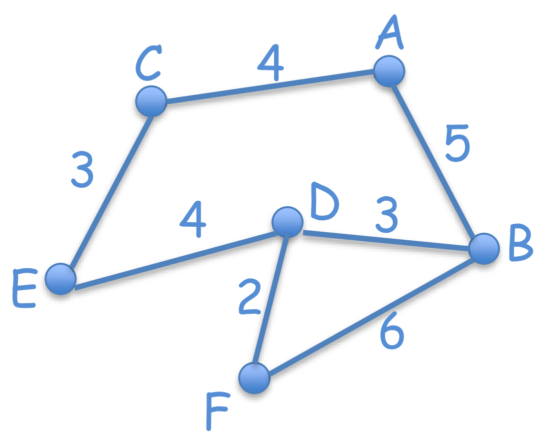
```
MST-Kruskal(G)

A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing order by
weight
for each edge (u,v)
     if Find-Set(u) ≠ Find-Set(v)
         A = A ∪ {u,v}
         Union(u,v)
return A
```



Kruskal's Algorithm

MST-Kruskal(G)

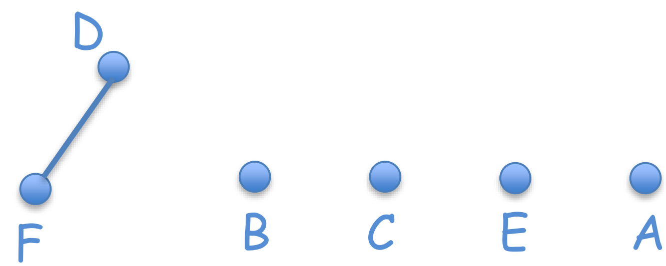
```
A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing order by
weight
for each edge (u,v)
    if Find-Set(u) ≠ Find-Set(v)
        A = A ∪ {u,v}
        Union(u,v)
return A
```

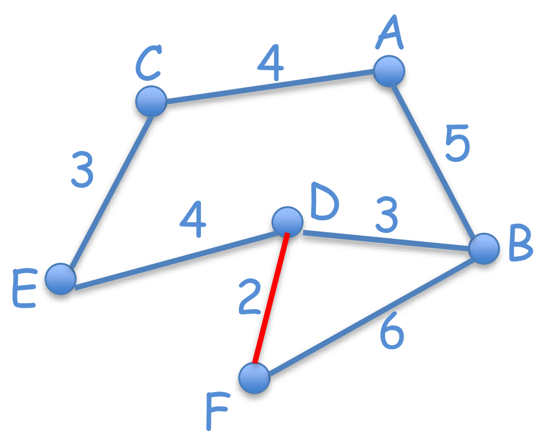


(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Kruskal's Algorithm

```
MST-Kruskal(G)

A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing order by
weight
for each edge (u,v)
     if Find-Set(u) ≠ Find-Set(v)
         A = A ∪ {u,v}
         Union(u,v)
return A
```



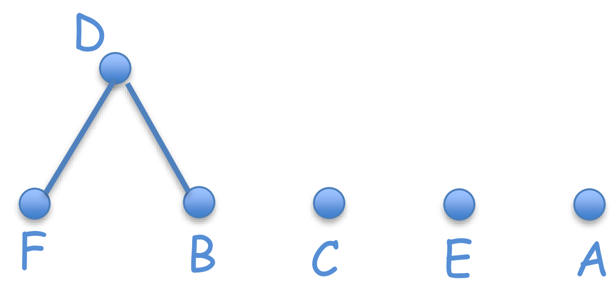


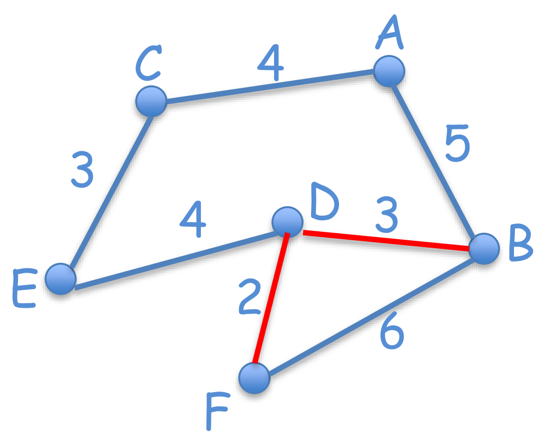
(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Kruskal's Algorithm

MST-Kruskal(G)

```
A = { }
for each v of V
     Make-Set(v)
sort all edges in increasing order by weight
for each edge (u,v)
     if Find-Set(u) ≠ Find-Set(v)
          A = A ∪ {u,v}
          Union(u,v)
return A
```



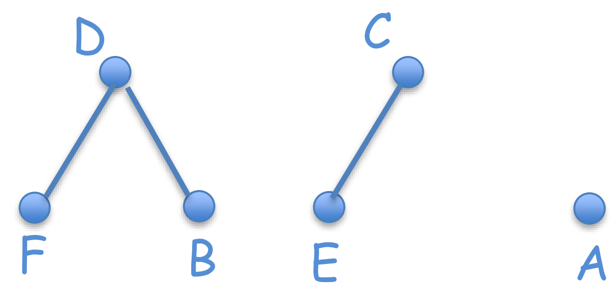


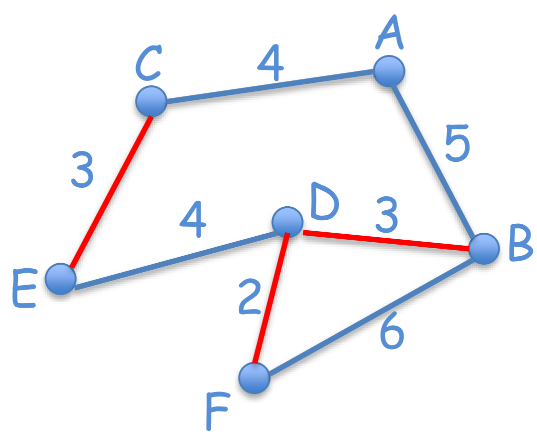
(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Kruskal's Algorithm

MST-Kruskal(G)

```
A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing order by
weight
for each edge (u,v)
    if Find-Set(u) ≠ Find-Set(v)
        A = A ∪ {u,v}
        Union(u,v)
return A
```



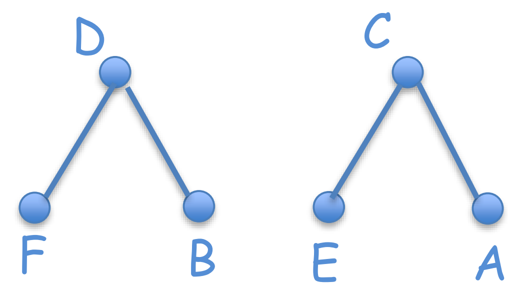


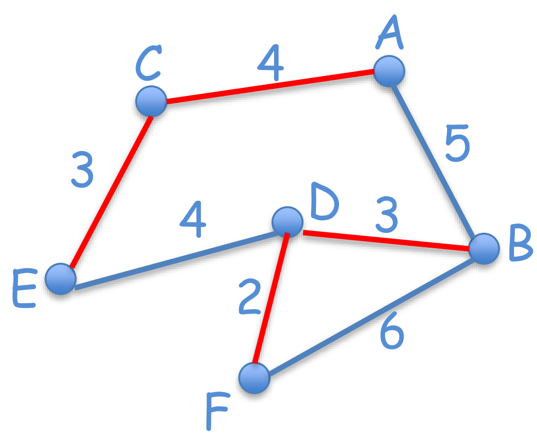
(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Kruskal's Algorithm

MST-Kruskal(G)

```
A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing order by
weight
for each edge (u,v)
    if Find-Set(u) ≠ Find-Set(v)
        A = A ∪ {u,v}
        Union(u,v)
return A
```



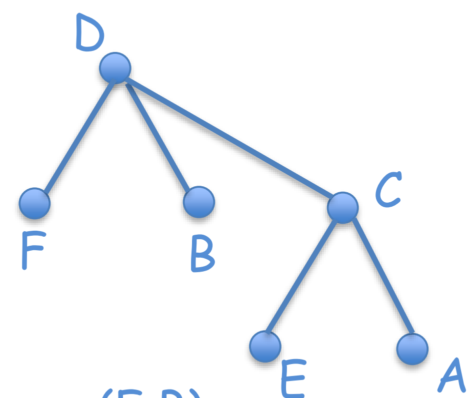


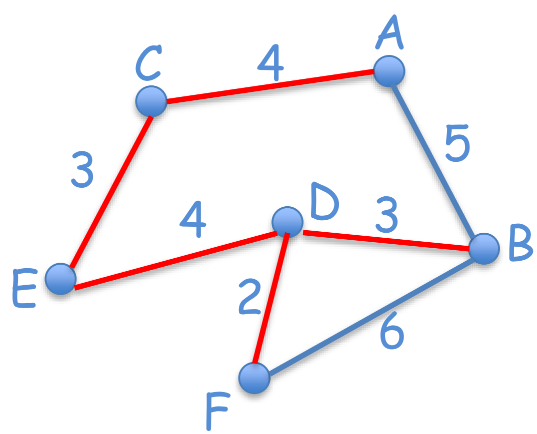
(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Kruskal's Algorithm

MST-Kruskal(G)

A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing order by weight
for each edge (u,v)
    if Find-Set(u) ≠ Find-Set(v)
        A = A ∪ {u,v}
        Union(u,v)
return A



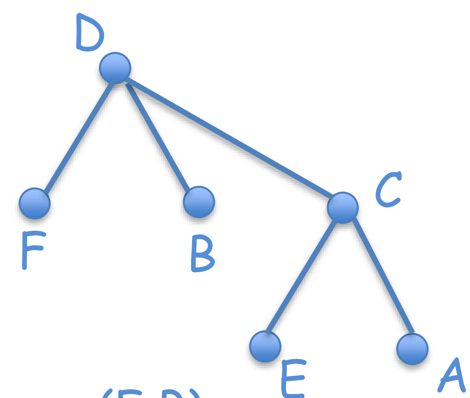


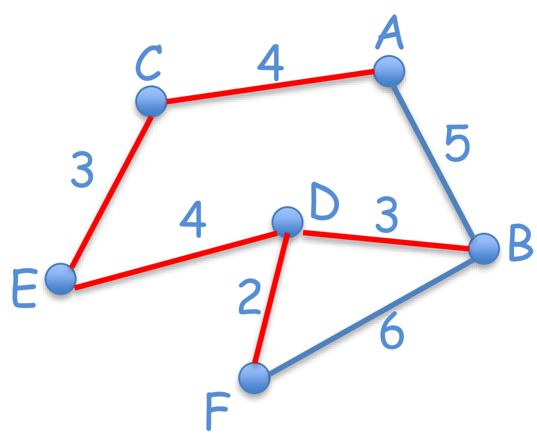
(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Kruskal's Algorithm

MST-Kruskal(G)

```
A = { }
for each v of V
     Make-Set(v)
sort all edges in increasing order by
weight
for each edge (u,v)
      if Find-Set(u) ≠ Find-Set(v)
          A = A ∪ {u,v}
          Union(u,v)
return A
```



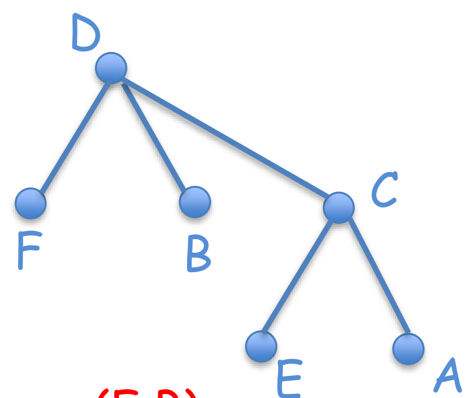


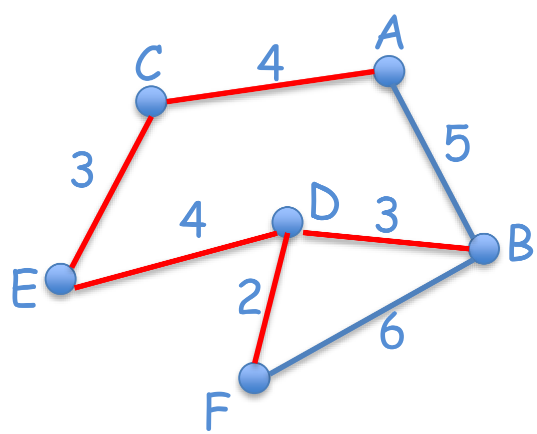
(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Kruskal's Algorithm

MST-Kruskal(G)

A = { }
for each v of V
    Make-Set(v)
sort all edges in increasing order by weight
for each edge (u,v)
    if Find-Set(u) ≠ Find-Set(v)
        A = A ∪ {u,v}
        Union(u,v)
return A





(F,B)
(A,B)
(D,E)
(A,C)
(C,E)
(D,B)
(D,F)

Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
     u.key = ∞
     u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
     u = ExtractMin(Q)
     for each v of Adj(u)
          if v in Q and w(u,v) < v.key
               v.par = u
               v.key = w(u,v)
```

Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
     u.key = ∞
     u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
     u = ExtractMin(Q)
     for each v of Adj(u)
          if v in Q and w(u,v) < v.key
               v.par = u
               v.key = w(u,v)
```

$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\} O(|V|)$

Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```

O(|V|)

O(|V|)

Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
    u.key = ∞                           O(|V|)
    u.par = nil
s.key = 0
create a minimum priority Q on V        O(|V|)
while Q ≠ { }
    u = ExtractMin(Q)                   O(|V|.log|V|)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```

Prim's Algorithm

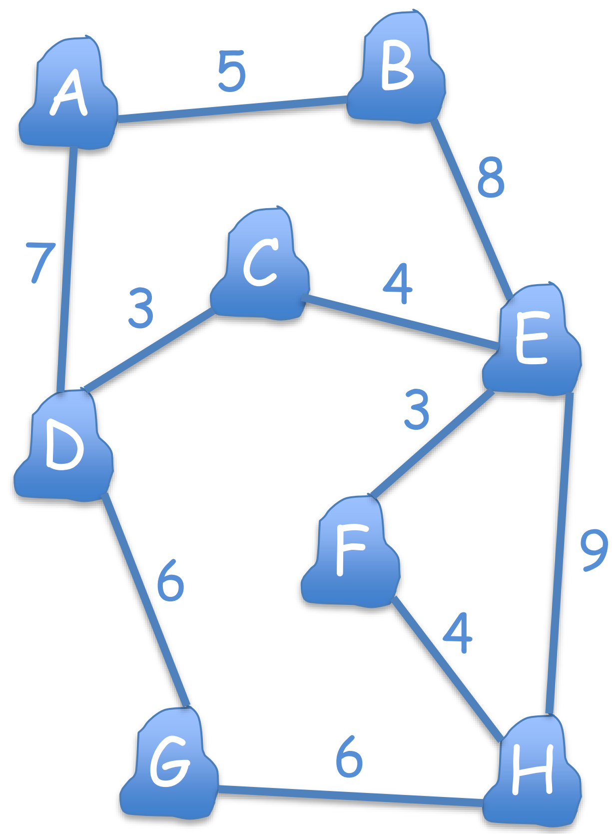
MST-Prim(G,s)

```
for each u of V                         } O(|V|)
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V        } O(|V|)
while Q ≠ { }
    u = ExtractMin(Q)                   → O(|V|.log|V|)
    for each v of Adj(u)                } O(|E|.log|V|)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```

Prim's Algorithm

MST-Prim(G,s)

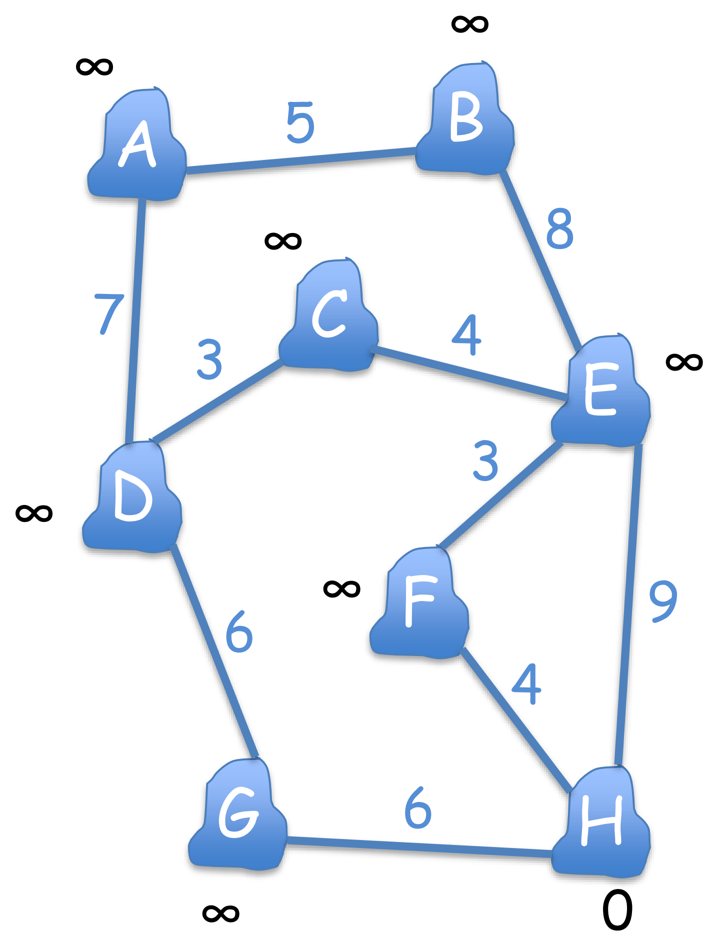
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```

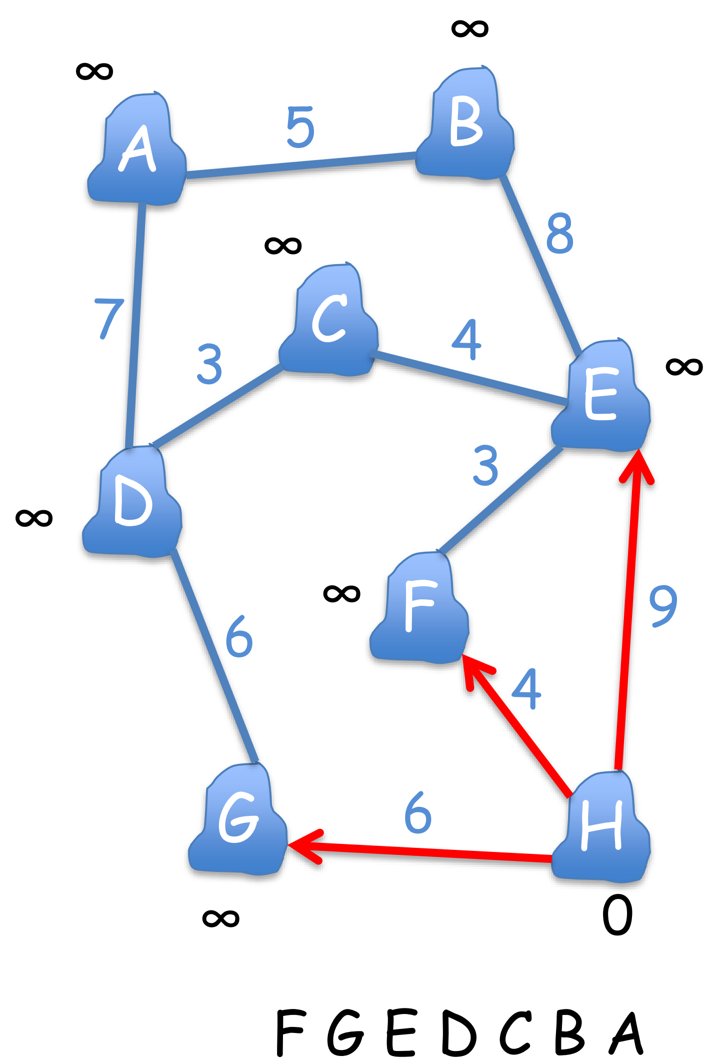


H F G E D C B A

Prim's Algorithm

MST-Prim(G,s)

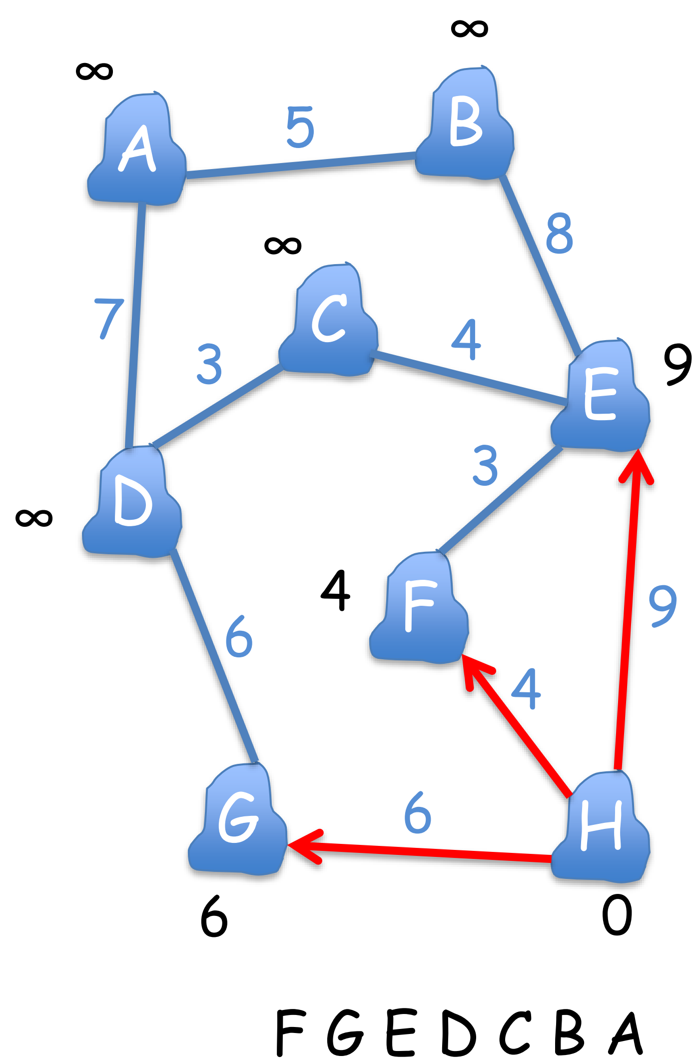
```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

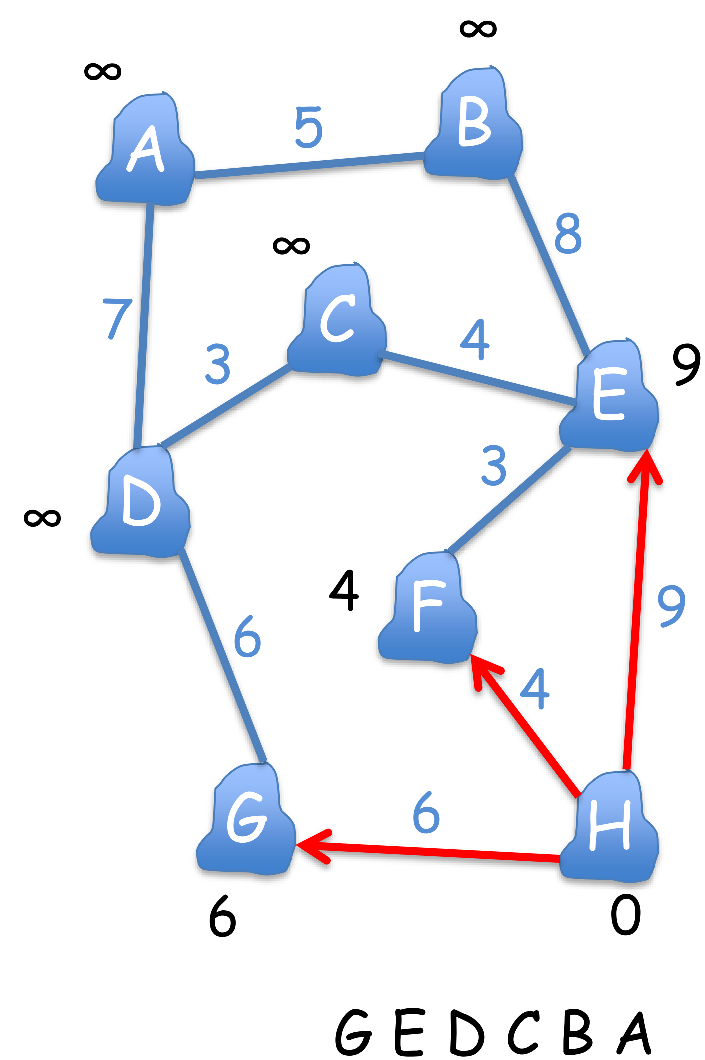
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

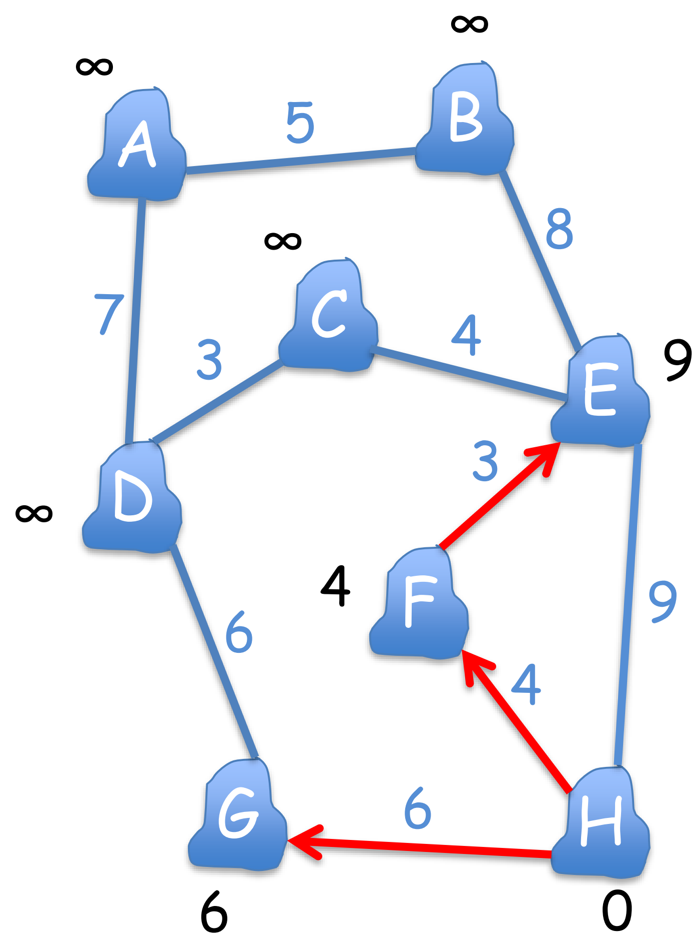
```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```

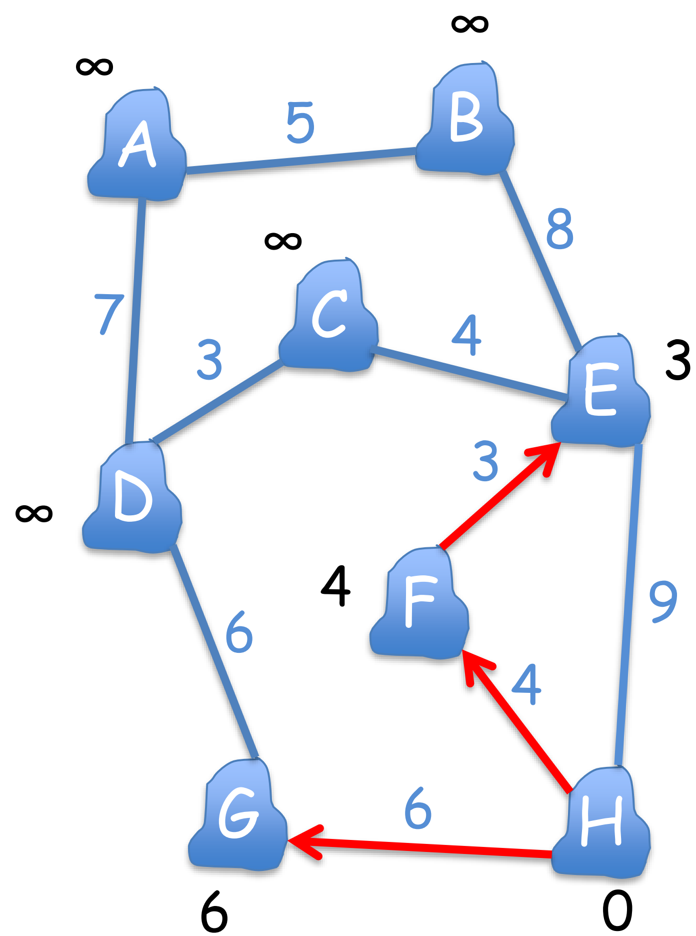


G E D C B A

Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```

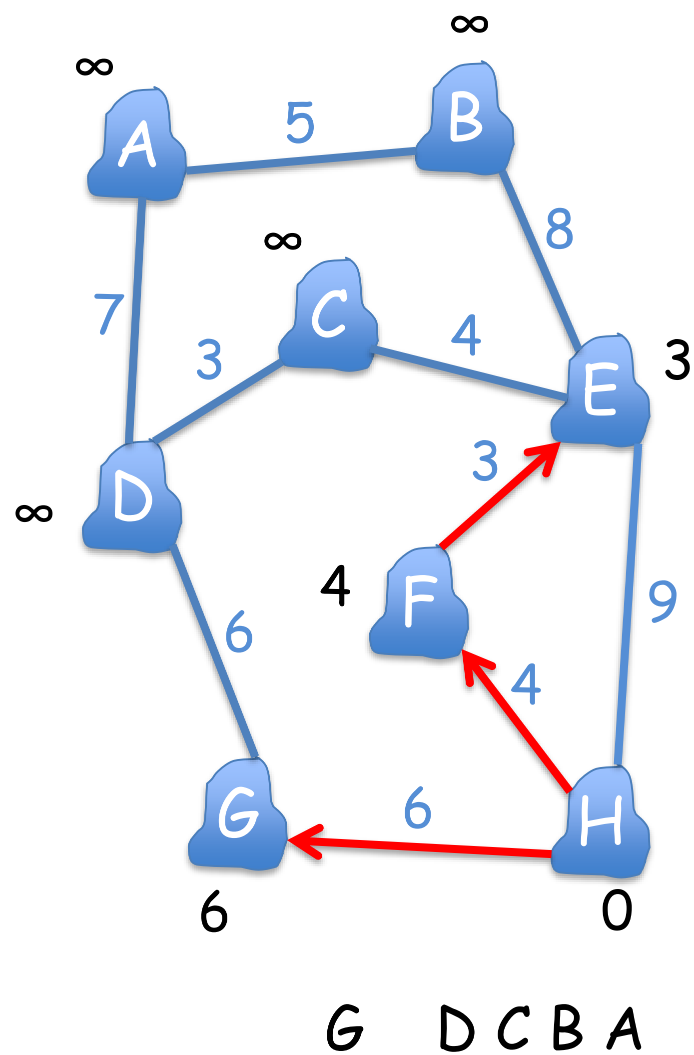


G E D C B A

Prim's Algorithm

MST-Prim(G,s)

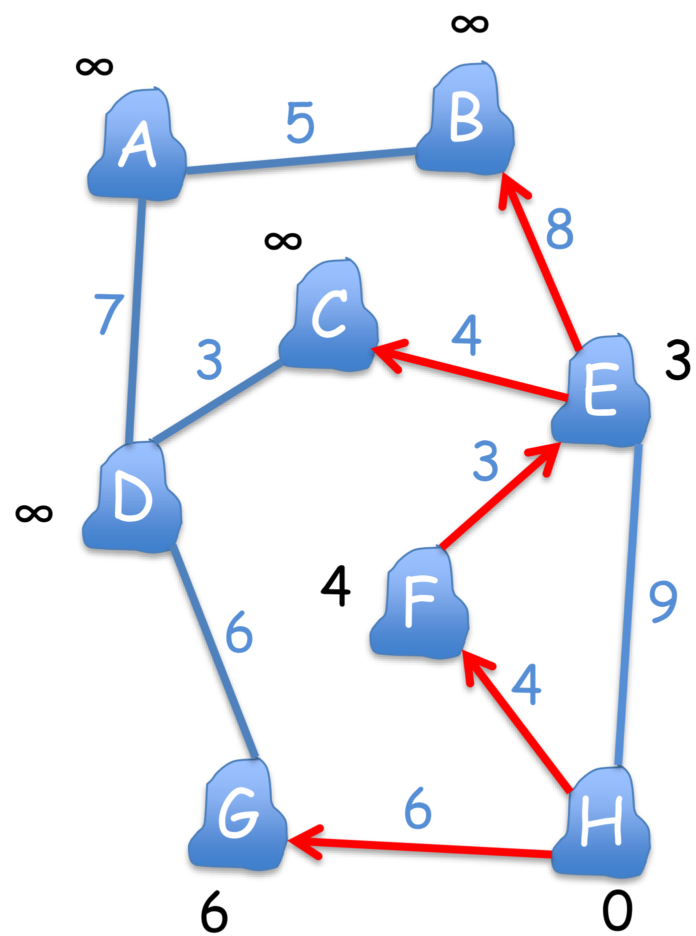
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

<u>MST-Prim(G,s)</u>

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```

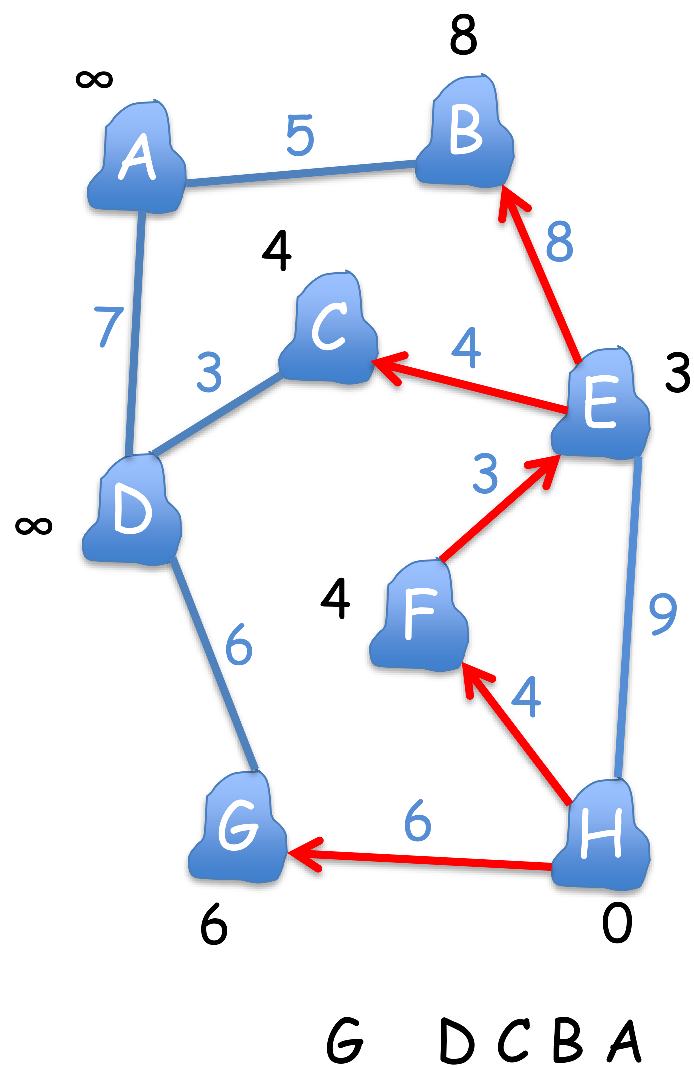


G D C B A

Prim's Algorithm

MST-Prim(G,s)

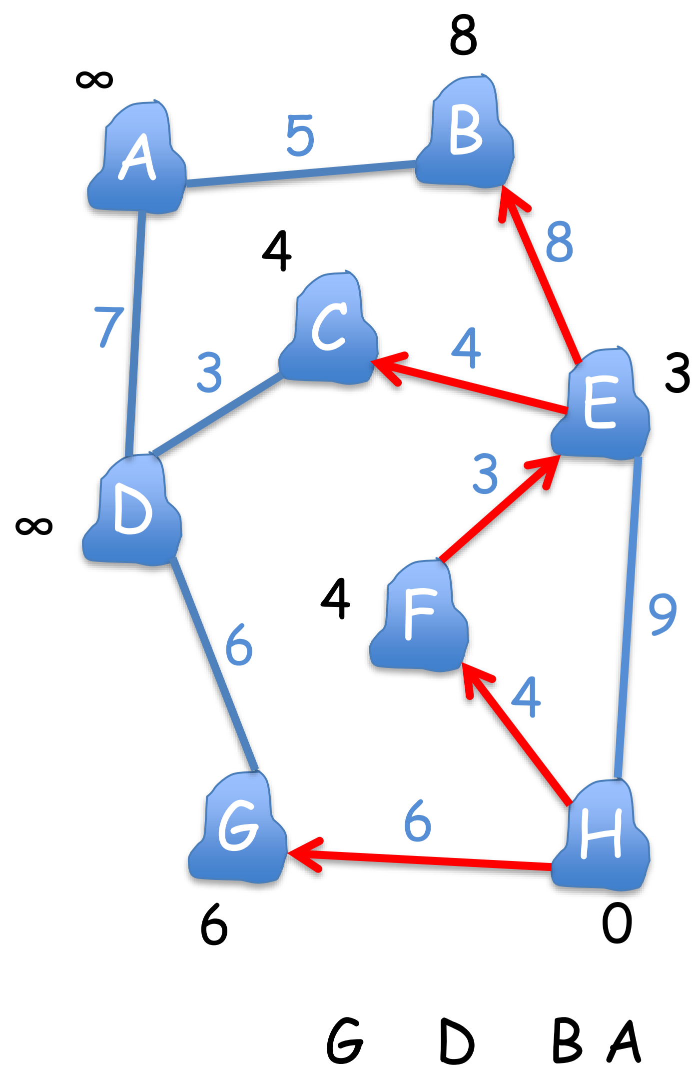
```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

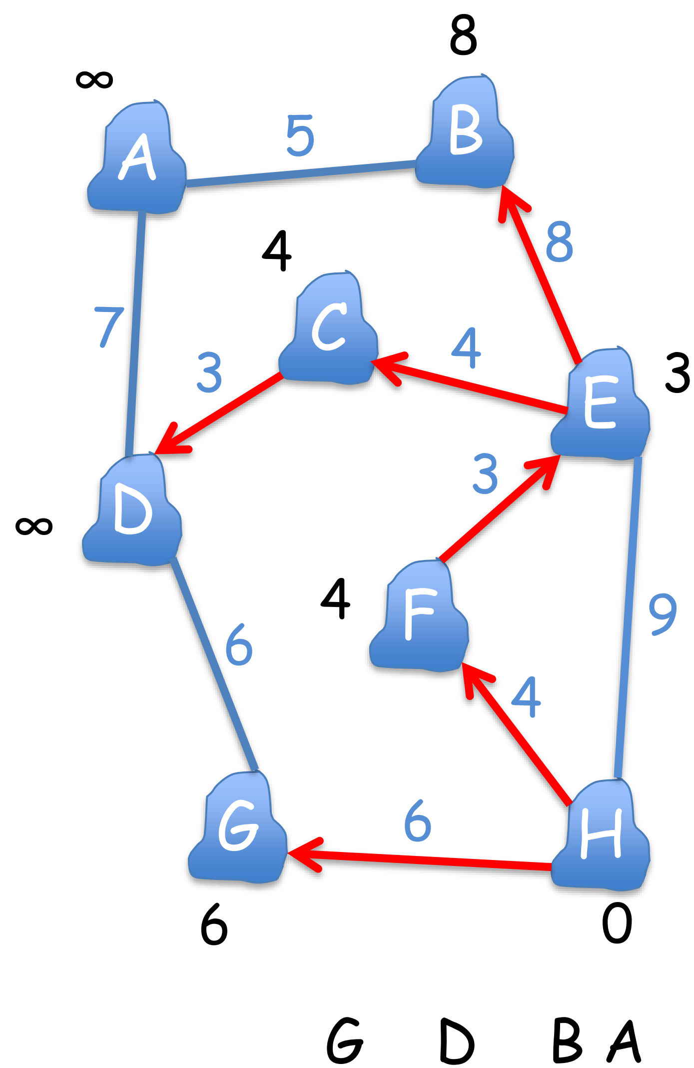
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

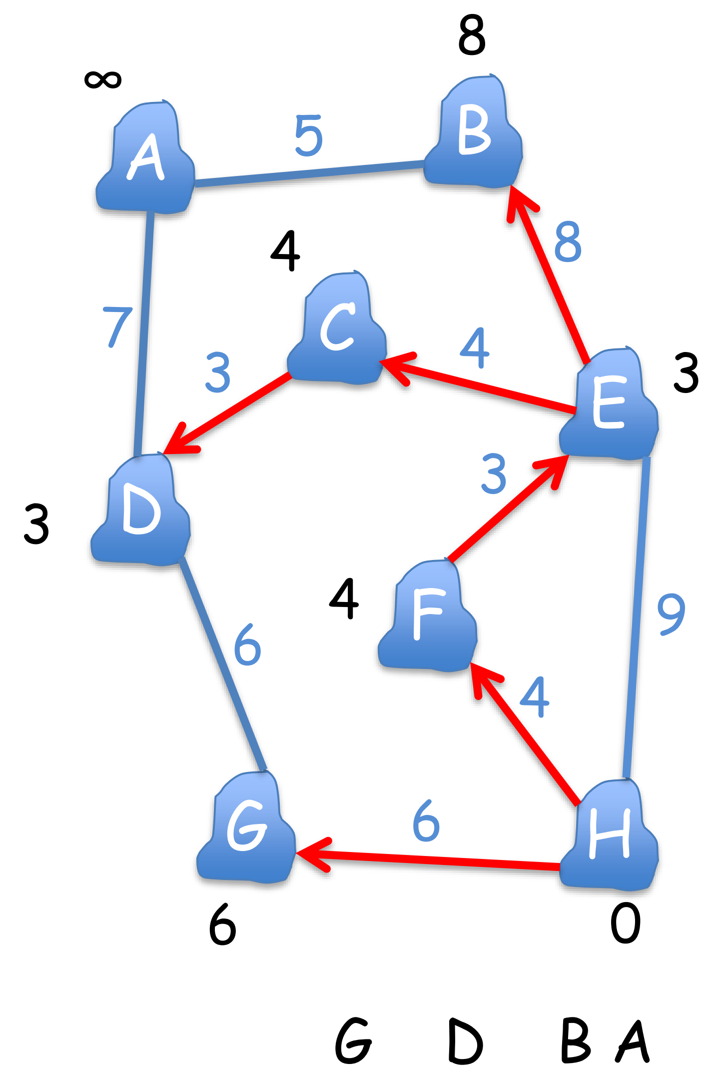
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

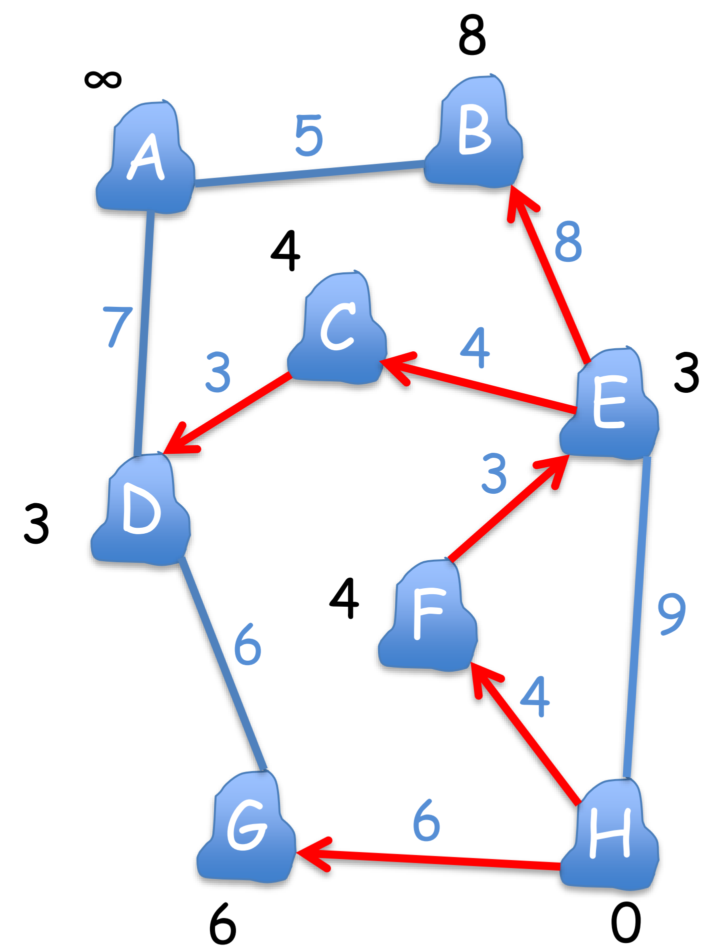
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```

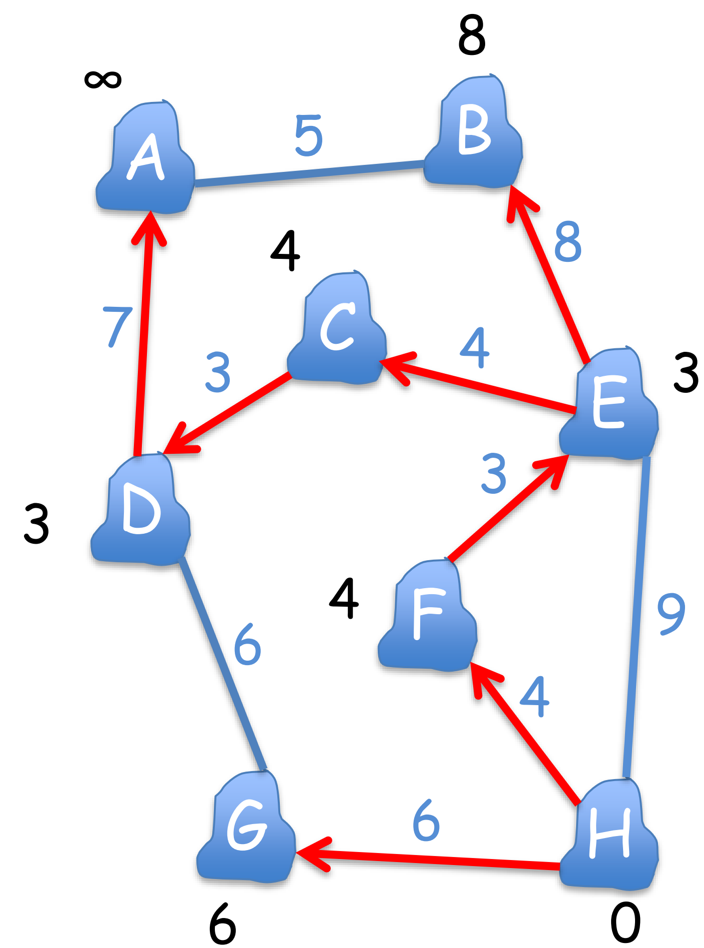


G B A

Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```

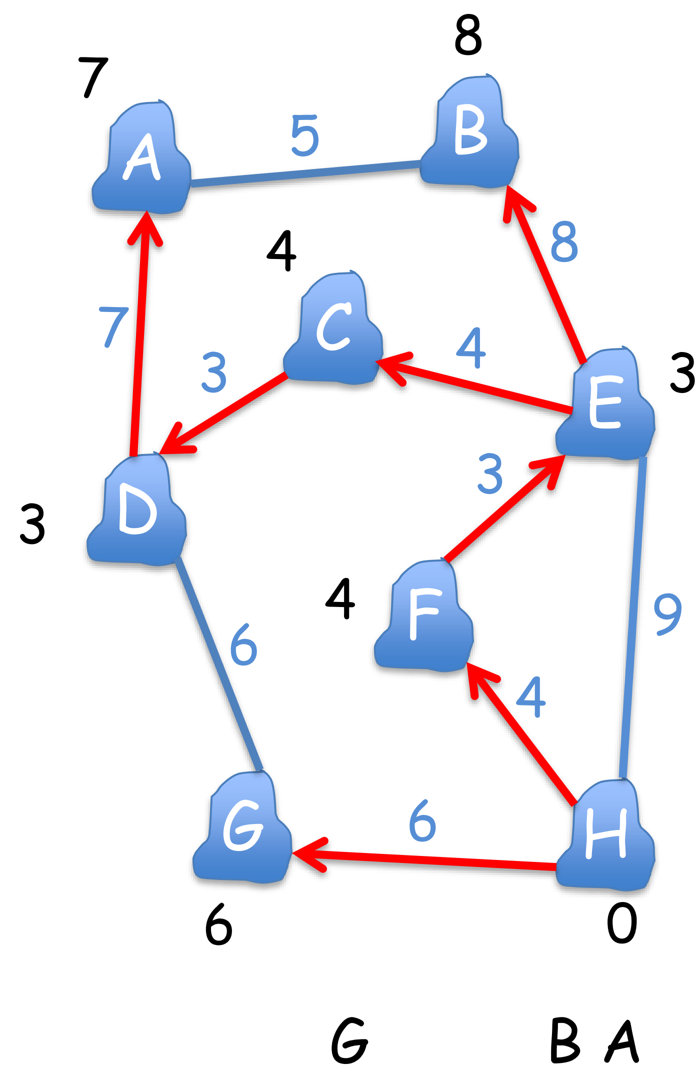


G B A

Prim's Algorithm

MST-Prim(G,s)

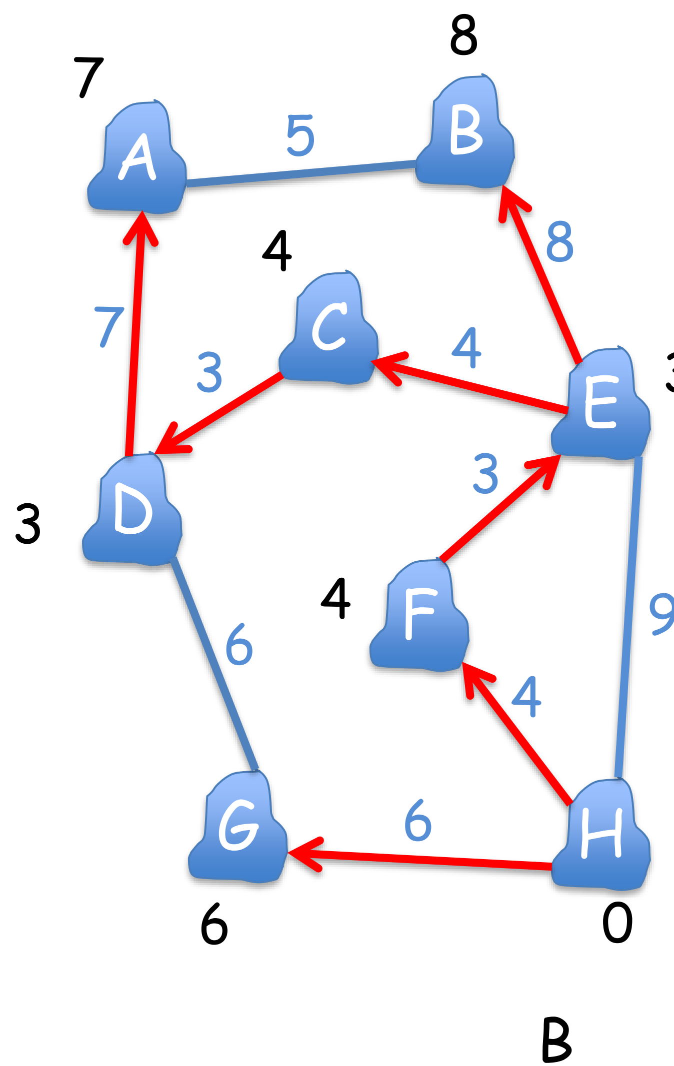
```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

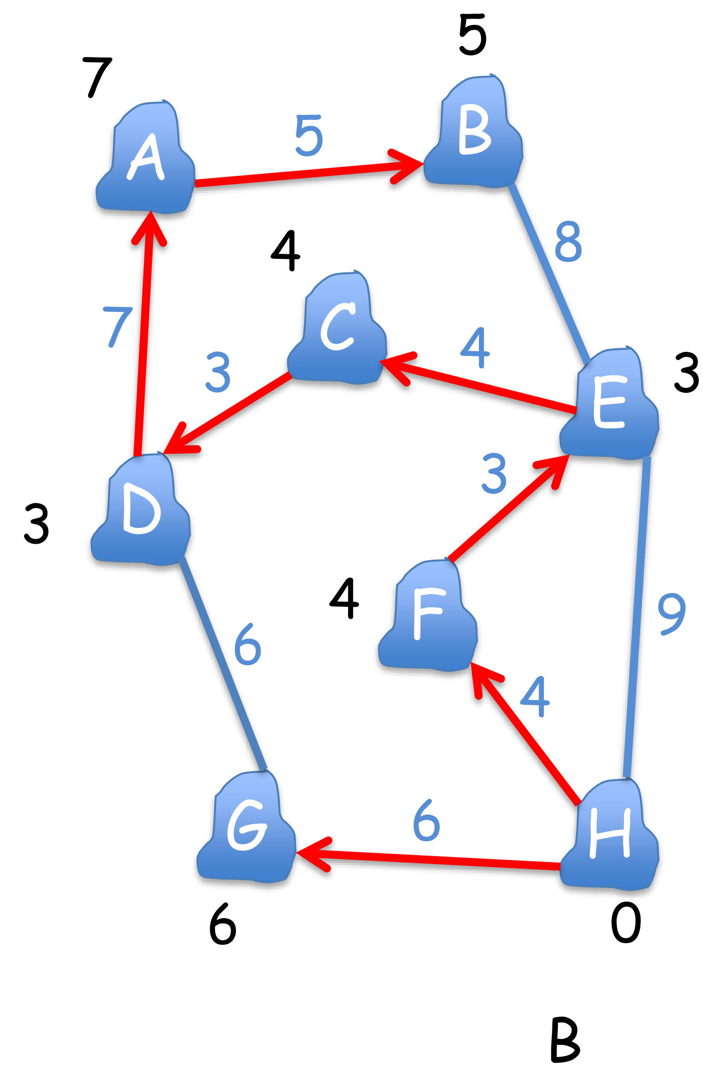
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

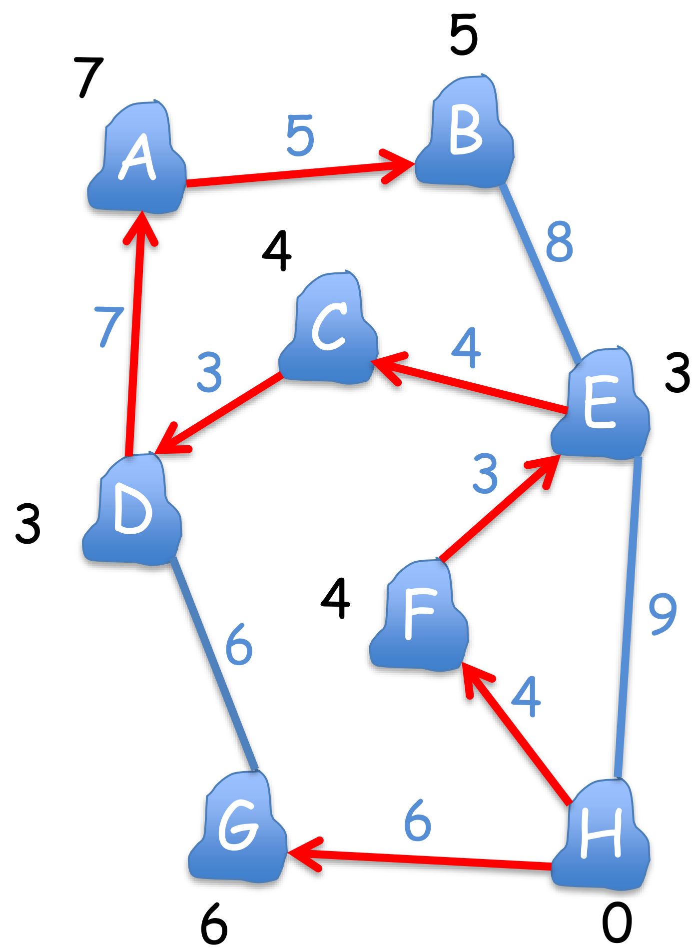
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

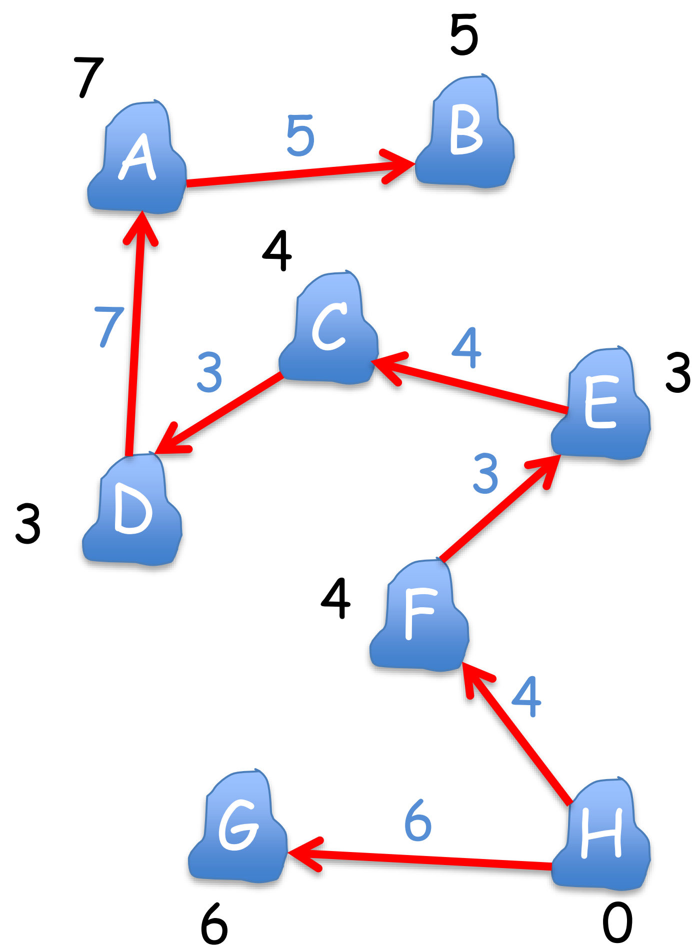
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      for each v of Adj(u)
            if v in Q and w(u,v) < v.key
                  v.par = u
                  v.key = w(u,v)
```



Prim's Algorithm

MST-Prim(G,s)

```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    for each v of Adj(u)
        if v in Q and w(u,v) < v.key
            v.par = u
            v.key = w(u,v)
```

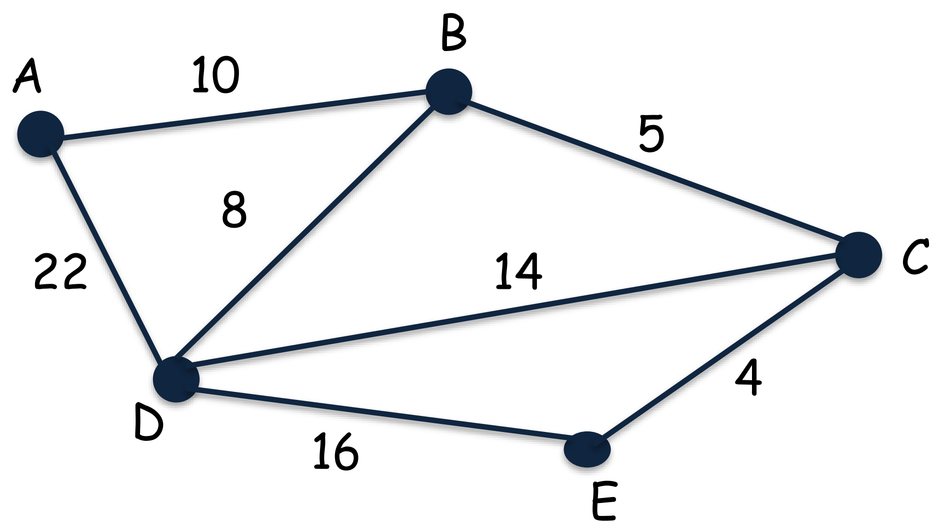


SSSP

- given a weighted graph $G=(V,E)$ and a source vertex s in V, find the shortest path from s to every other vertex in V

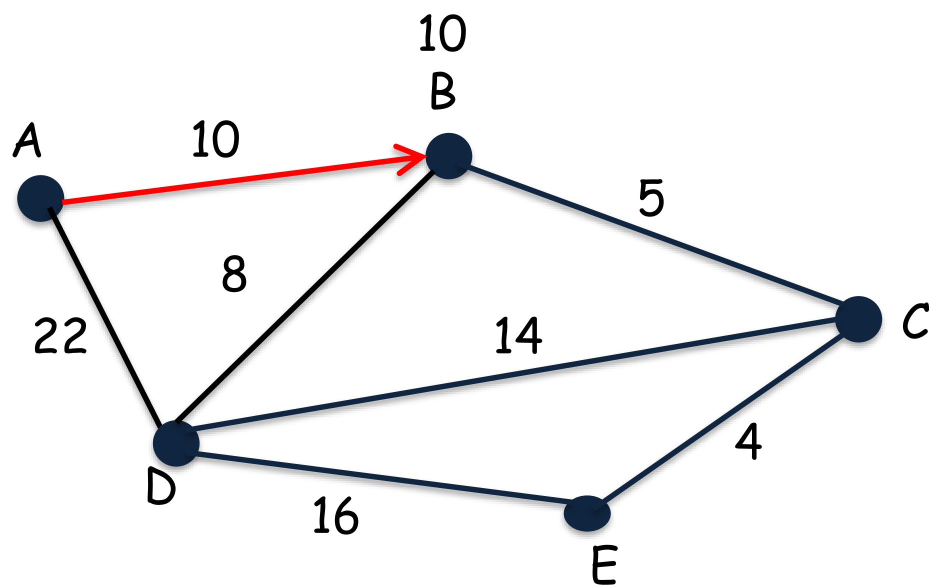
SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



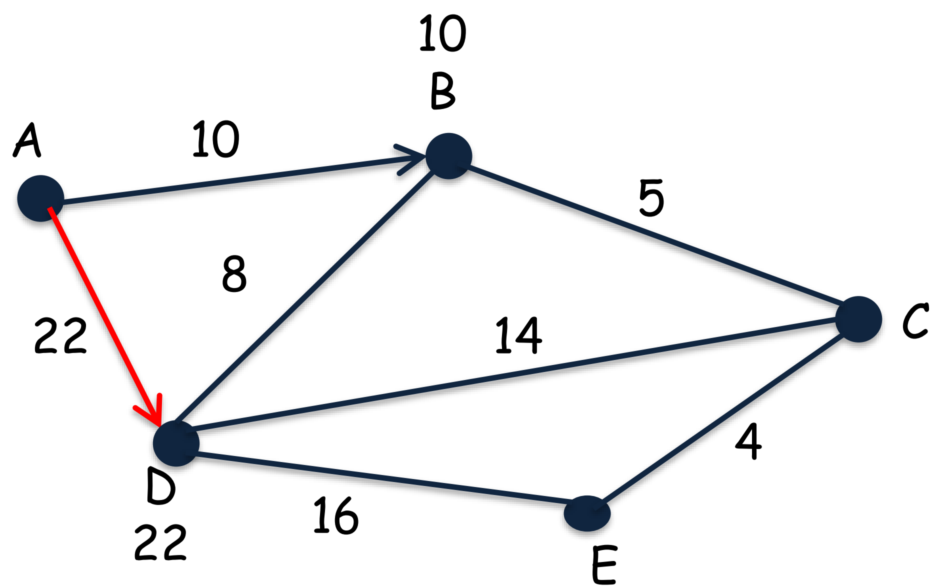
SSSP

- given a weighted graph $G=(V,E)$ and a source vertex s in V, find the shortest path from s to every other vertex in V



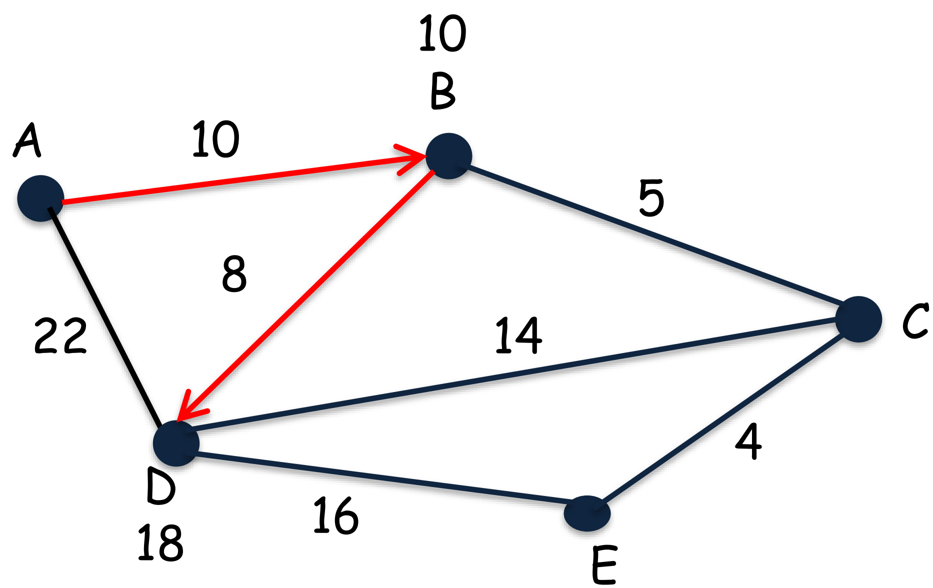
SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



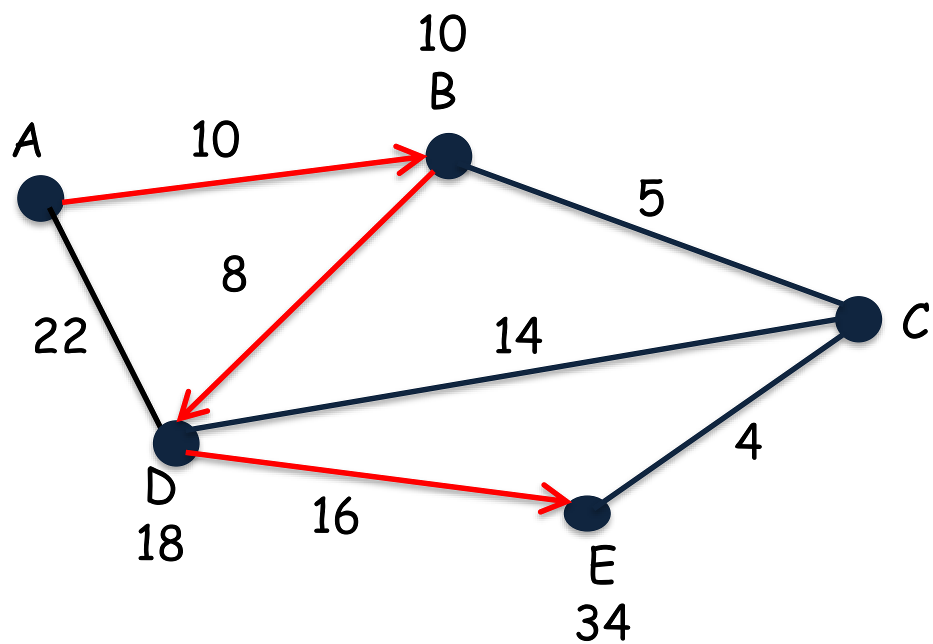
SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



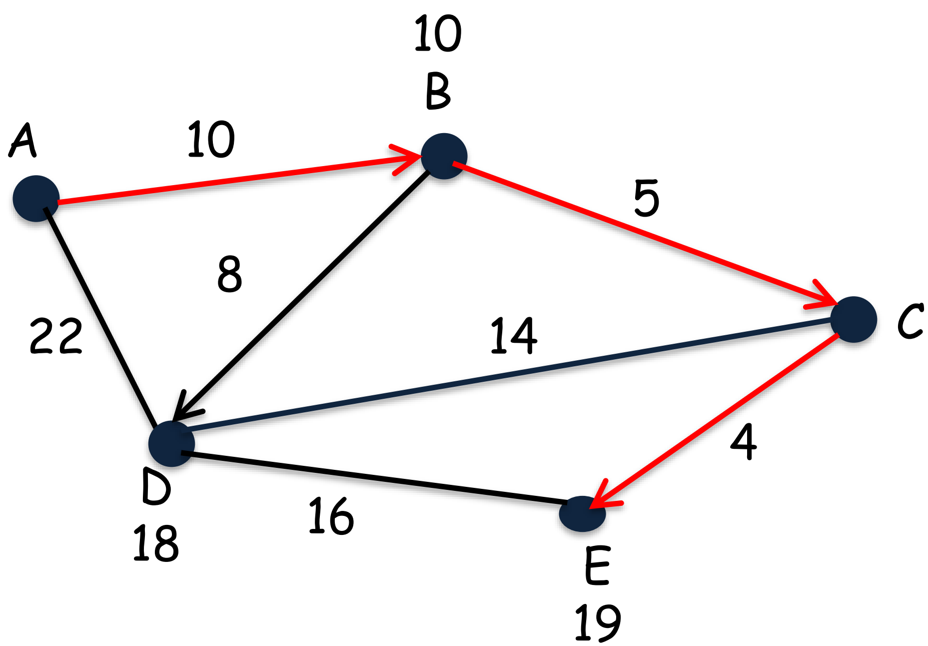
SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



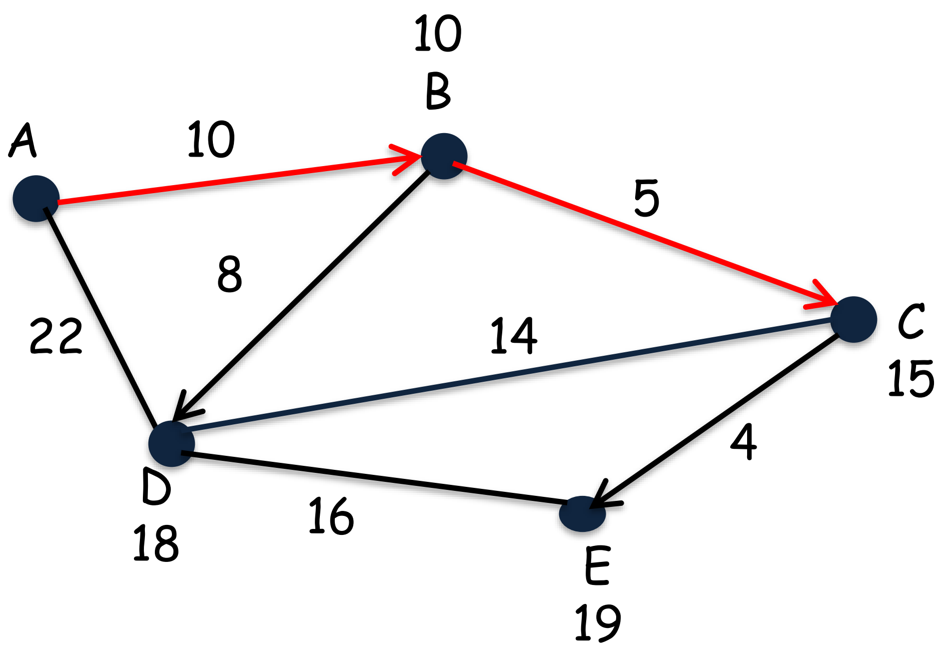
SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



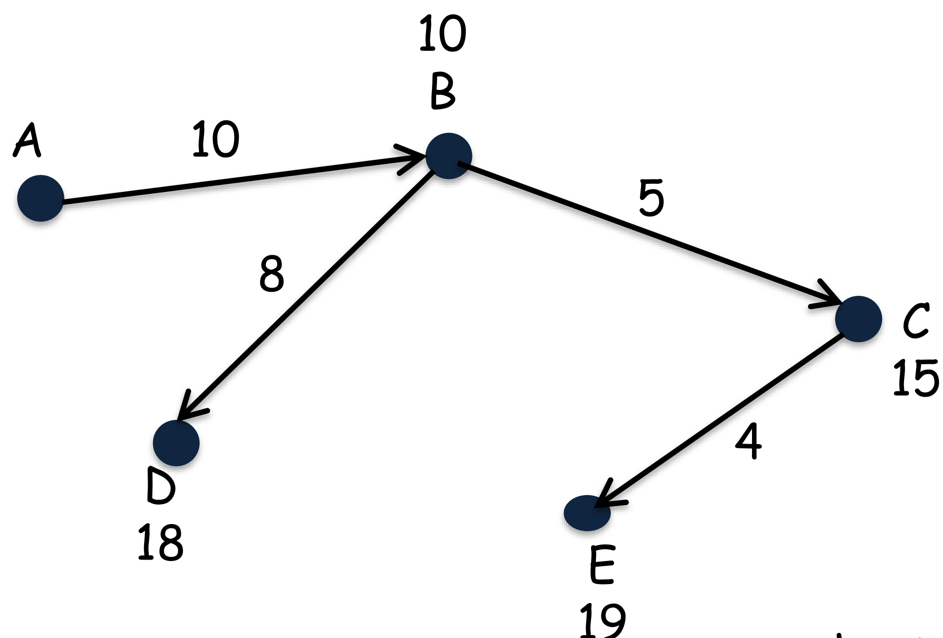
SSSP

- given a weighted graph $G=(V,E)$ and a source vertex s in V, find the shortest path from s to every other vertex in V



SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V



shortest-paths tree

SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V
- the weight of each edge fixed as 1

SSSP

- given a weighted graph $G=(V,E)$ and a source vertex s in V, find the shortest path from s to every other vertex in V

- the weight of each edge fixed as 1

--BFS--

SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V

- the weight of each edge fixed as 1

--BFS--

- the weight of each edge not fixed but non-negative

SSSP

- given a weighted graph G=(V,E) and a source vertex s in V, find the shortest path from s to every other vertex in V

- the weight of each edge fixed as 1

--BFS--

- the weight of each edge not fixed but non-negative

--Dijkstra—

Relaxation

- For each vertex v in V, initialize two parameters :

Relaxation

- For each vertex v in V, initialize two parameters :
  - parent pointer – indicates the predecessor of the vertex in the shortest path from s to v

Relaxation

- For each vertex v in V, initialize two parameters :
  - parent pointer – indicates the predecessor of the vertex in the shortest path from s to v
  - distance – indicates the shortest-path estimate from vertex to the source

Relaxation

- For each vertex v in V, initialize two parameters :
  - parent pointer – indicates the predecessor of the vertex in the shortest path from s to v
  - distance – indicates the shortest-path estimate from vertex to the source

Initialize (G, s)

```
for each vertex v i V
     v.dis = ∞
     v.par = nil
s.dis = 0
```

Relaxation

- relaxing an edge (u,v) : testing whether the shortest path to the vertex v can be improved by going through the vertex u

Relaxation

- relaxing an edge (u,v) : testing whether the shortest path to the vertex v can be improved by going through the vertex u

Relax(u, v)

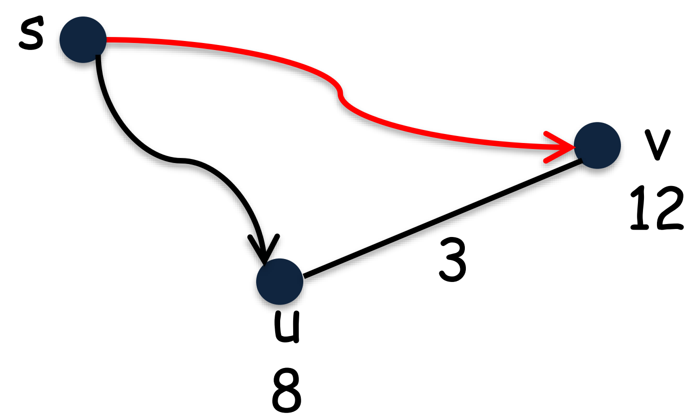
```
if  v.dis > u.dis + w(u,v)
      v.dis = u.dis + w(u,v)
      v.par = u
```

Relaxation

- relaxing an edge (u,v) : testing whether the shortest path to the vertex v can be improved by going through the vertex u

Relax(u, v)

```
if  v.dis > u.dis + w(u,v)
     v.dis = u.dis + w(u,v)
     v.par = u
```

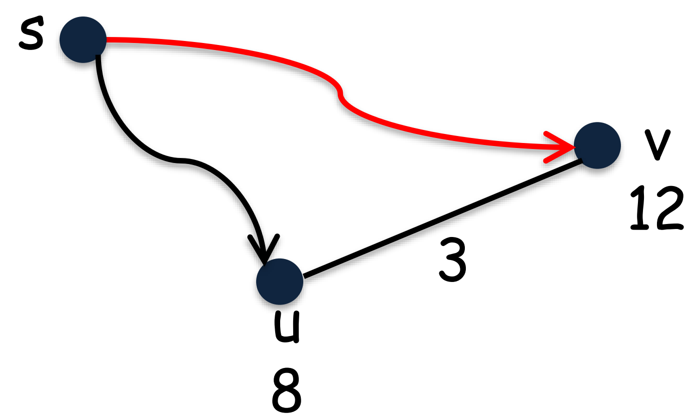


Relaxation

- relaxing an edge (u,v) : testing whether the shortest path to the vertex v can be improved by going through the vertex u

Relax(u, v)

if v.dis > u.dis + w(u,v)
    v.dis = u.dis + w(u,v)
    v.par = u



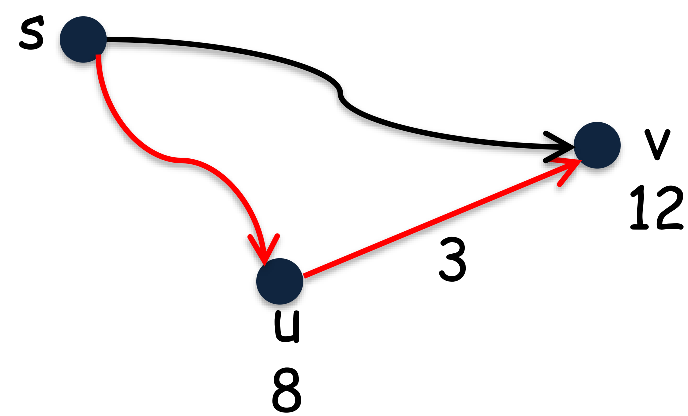
v.dis > u.dis + w(u,v)
12 > 8 + 3

Relaxation

- relaxing an edge (u,v) : testing whether the shortest path to the vertex v can be improved by going through the vertex u

Relax(u, v)

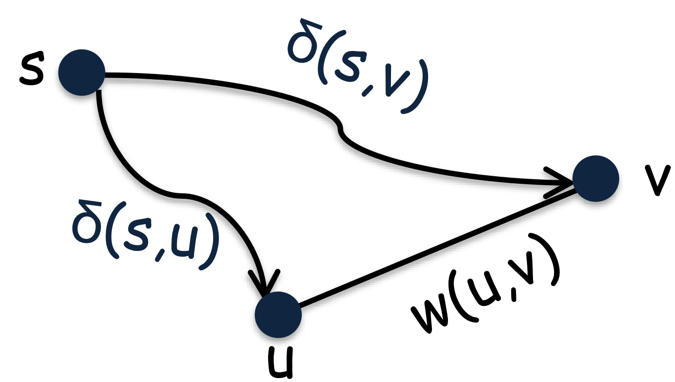
if v.dis > u.dis + w(u,v)
  v.dis = u.dis + w(u,v)
  v.par = u



v.dis > u.dis + w(u,v) → v.dis = u.dis + w(u,v)
12 > 8 + 3
v.dis = 11
v.par = u

Relaxation

- Let δ(s,v) be the weight of the shortest path from source to the vertex v (after the termination of the program)

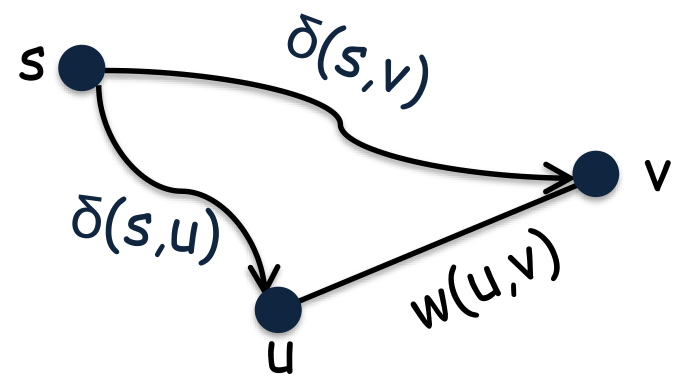


Relaxation

- Let $\delta(s,v)$ be the weight of the shortest path from source to the vertex v (after the termination of the program)

- For any edge (u,v) in E,

$$\delta(s,v) \leq \delta(s,u) + w(u,v)$$



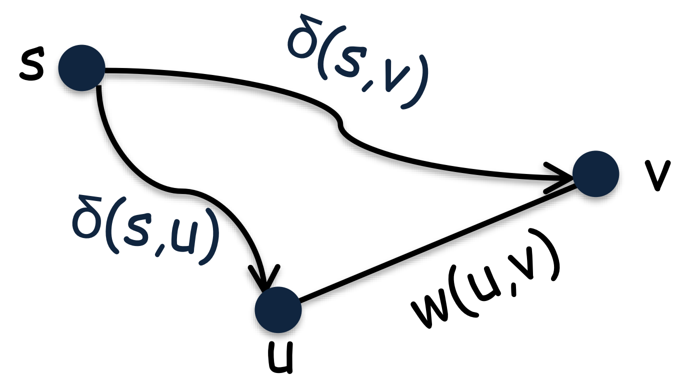
Relaxation

- Let $\delta(s,v)$ be the weight of the shortest path from source to the vertex v (after the termination of the program)
- For any edge (u,v) in E,

$$\delta(s,v) \leq \delta(s,u) + w(u,v)$$

- For all vertices v in V,

$$v.dis \geq \delta(s,v)$$

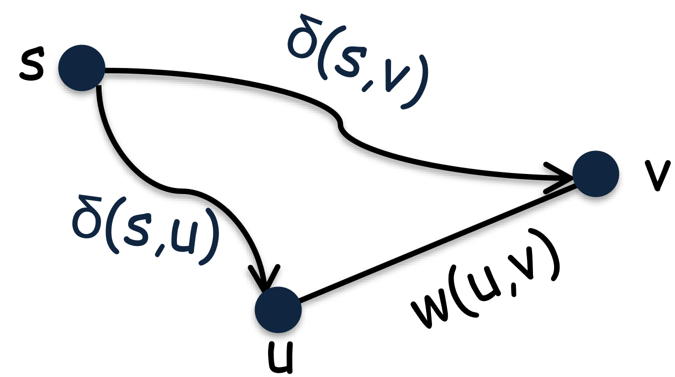


Relaxation

- Let $\delta(s,v)$ be the weight of the shortest path from source to the vertex v (after the termination of the program)

- For any edge (u,v) in E,

$$\delta(s,v) \leq \delta(s,u) + w(u,v)$$



- For all vertices v in V,

$$v.dis \geq \delta(s,v)$$

- If there is no path from s to v, then

$$v.dis = \delta(s,v) = \infty$$

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if  v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
         update Q
```

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
     u.key = ∞
     u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
     u = ExtractMin(Q)
     S = S ∪ {u}
     for each v of Adj(u)
          if  v.dis > u.dis + w(u,v)
               v.dis = u.dis + w(u,v)
               v.par = u
          update Q
```

Initialize(G,s)
O(|V|)

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
     u.key = ∞
     u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
     u = ExtractMin(Q)
     S = S ∪ {u}
     for each v of Adj(u)
          if  v.dis > u.dis + w(u,v)
               v.dis = u.dis + w(u,v)
               v.par = u
          update Q
```

Initialize(G,s) $O(|V|)$

$O(|V|)$

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V                          Initialize(G,s)
     u.key = ∞                           O(|V|)
     u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V         O(|V|)
while Q ≠ { }
     u = ExtractMin(Q)                   O(|V|.log|V|)
     S = S ∪ {u}
     for each v of Adj(u)
          if  v.dis > u.dis + w(u,v)
               v.dis = u.dis + w(u,v)
               v.par = u
          update Q
```

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
            update Q
```

Initialize(G,s)
O(|V|)

O(|V|)

O(|V|.log|V|)

Relax(u,v)
O(1)

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                v.dis = u.dis + w(u,v)
                v.par = u
           update Q
```

Initialize(G,s)
O(|V|)

O(|V|)

O(|V|.log|V|)

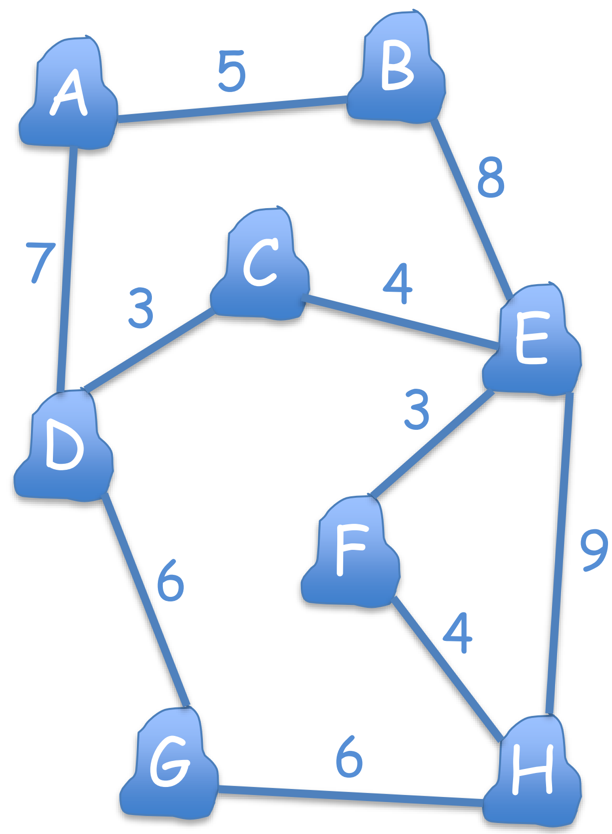
O(|E|.log|V|)

Relax(u,v)
O(1)

Dijkstra's Algorithm

Dijkstra(G,s)

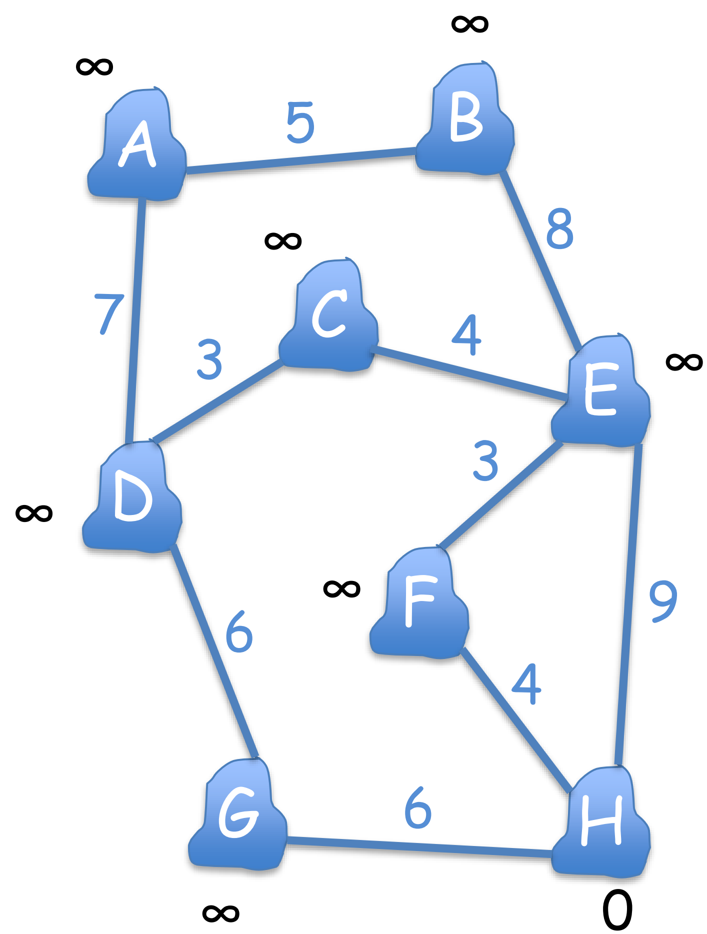
```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
            update Q
```



Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
           update Q
```



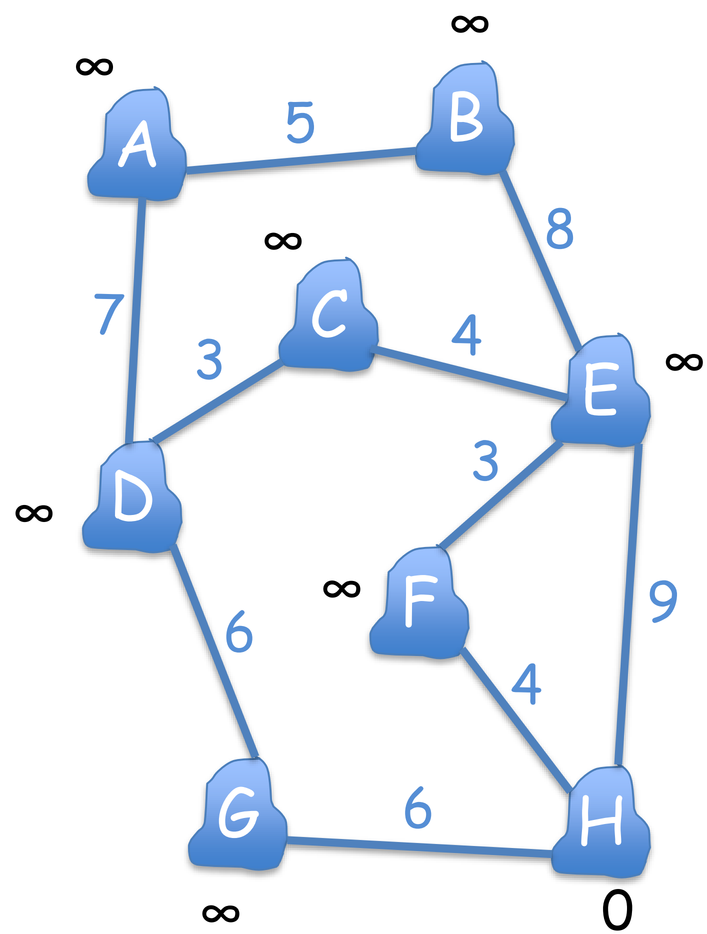
H F G E D C B A

S = { }

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                v.dis = u.dis + w(u,v)
                v.par = u
          update Q
```



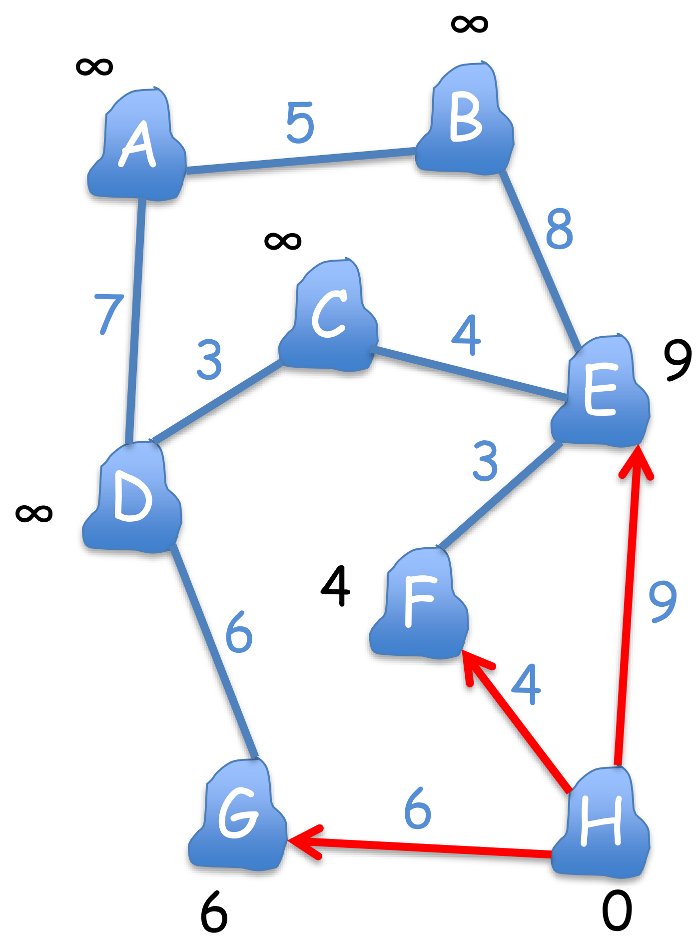
F G E D C B A

S = {H}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
           update Q
```



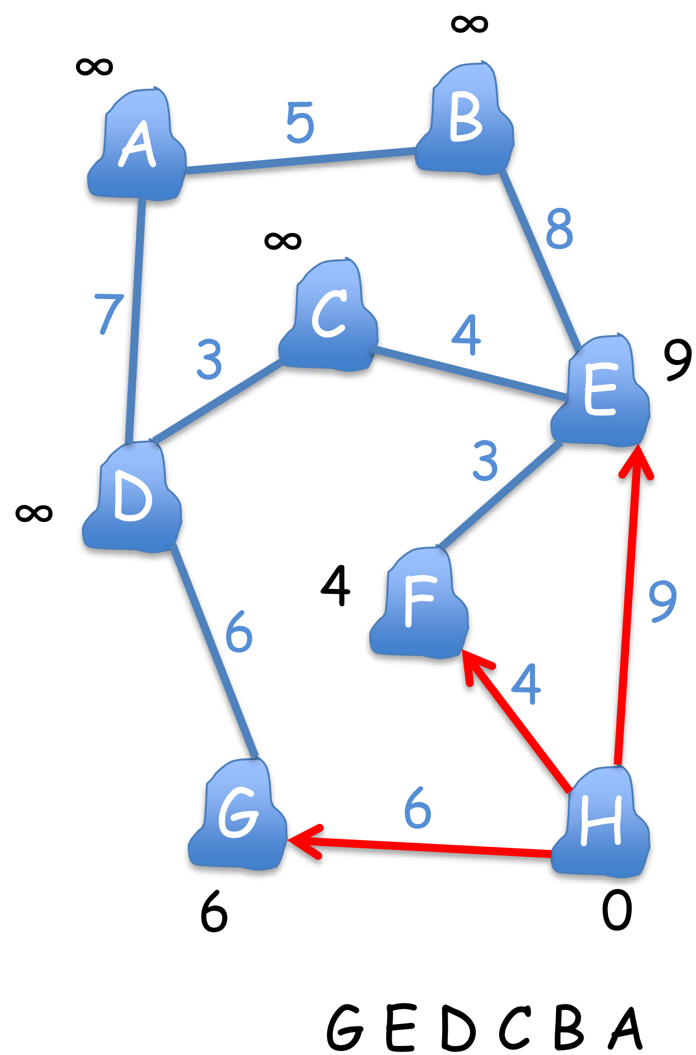
F G E D C B A

S = {H}

Dijkstra's Algorithm

<u>Dijkstra(G,s)</u>

```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if  v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
        update Q
```

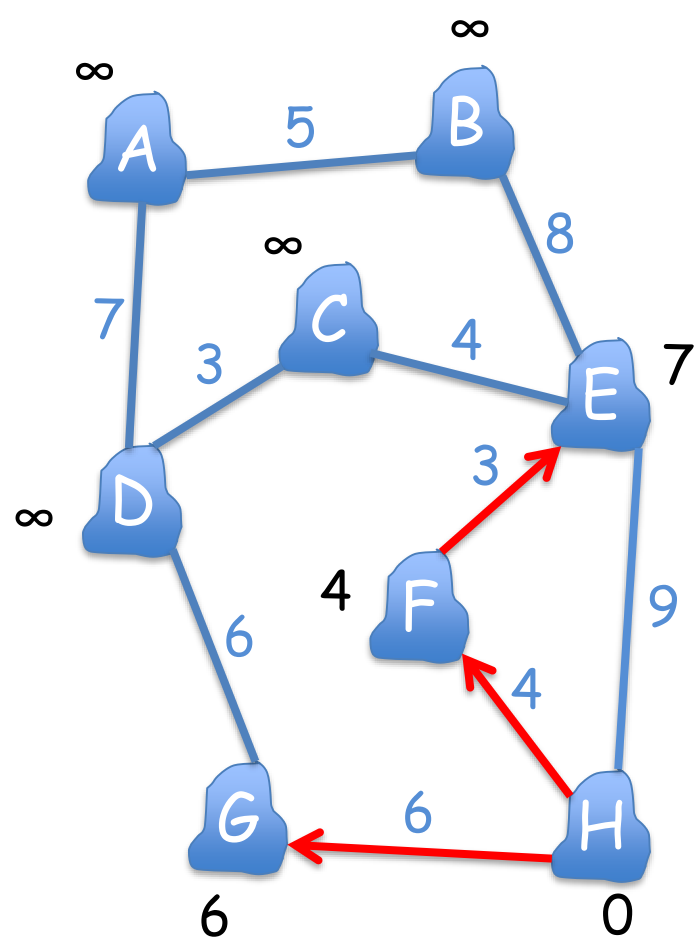


S = {H,F}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



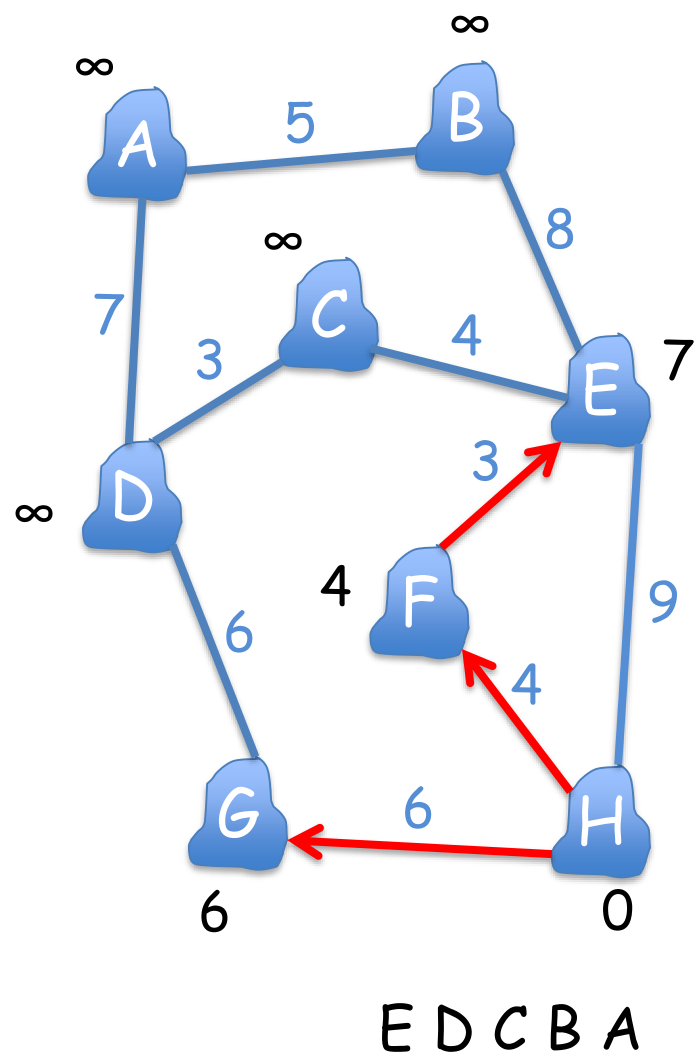
G E D C B A

S = {H,F}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if  v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
        update Q
```

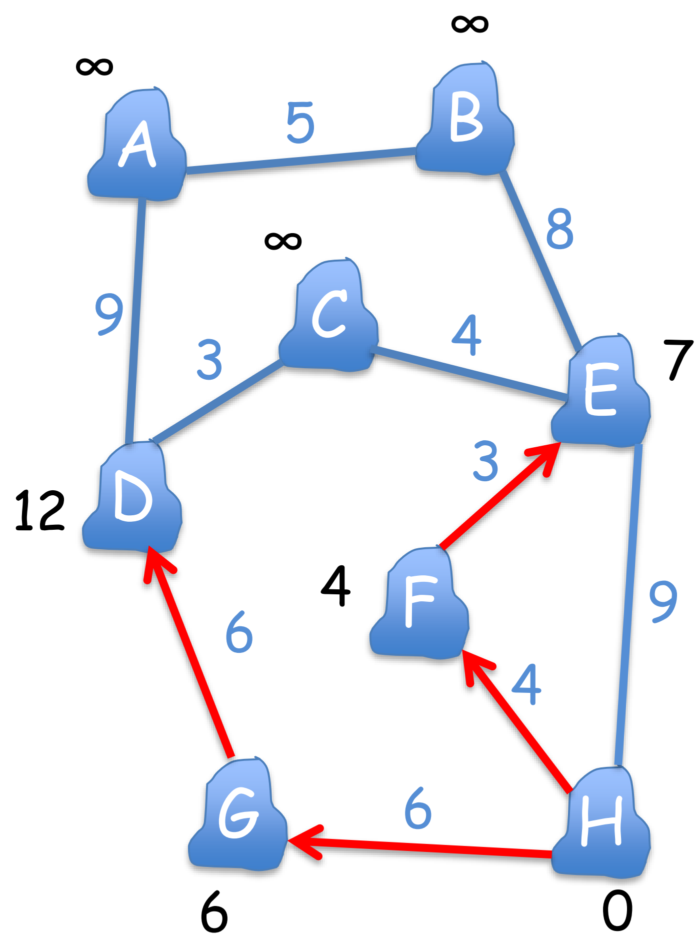


S = {H,F,G}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```

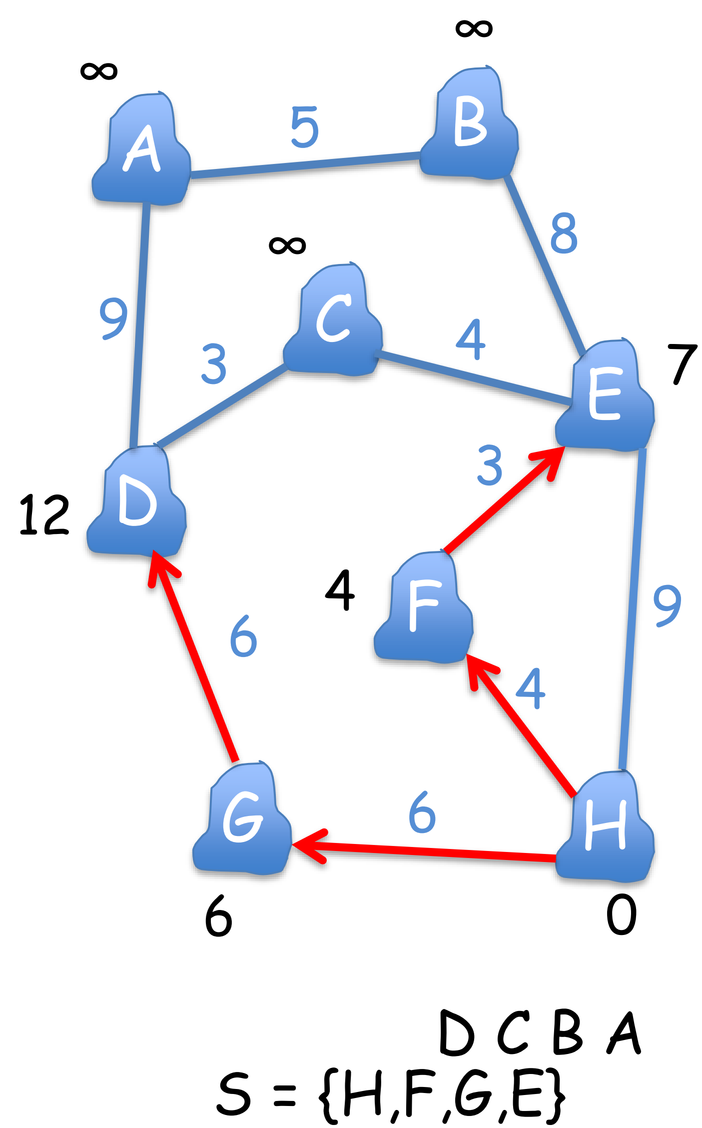


E D C B A

S = {H,F,G}

Dijkstra's Algorithm

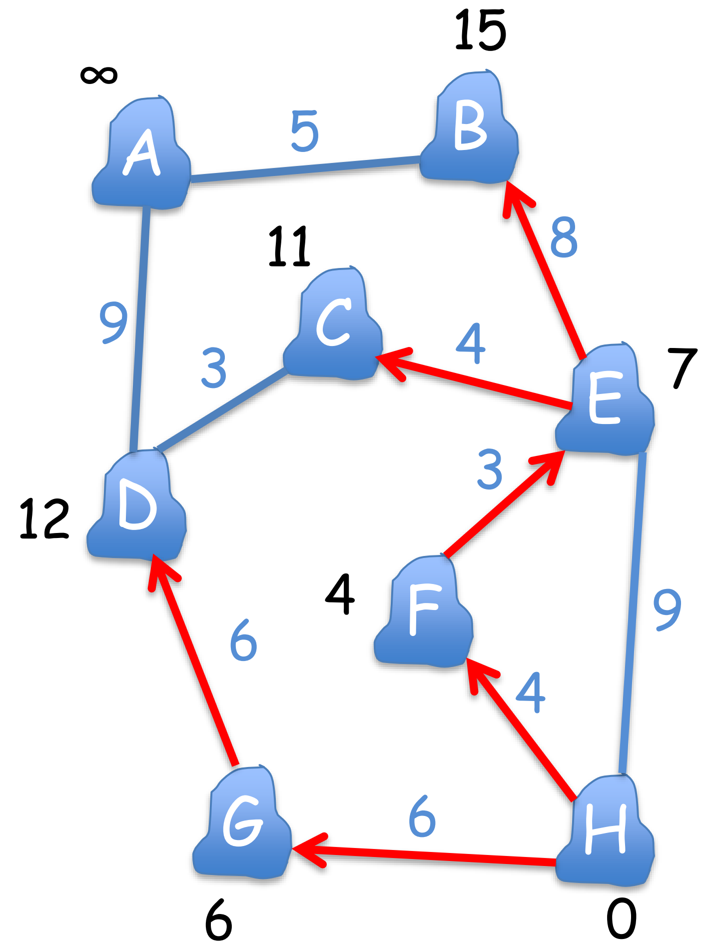
```
Dijkstra(G,s)

for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
            update Q
```



Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
            update Q
```

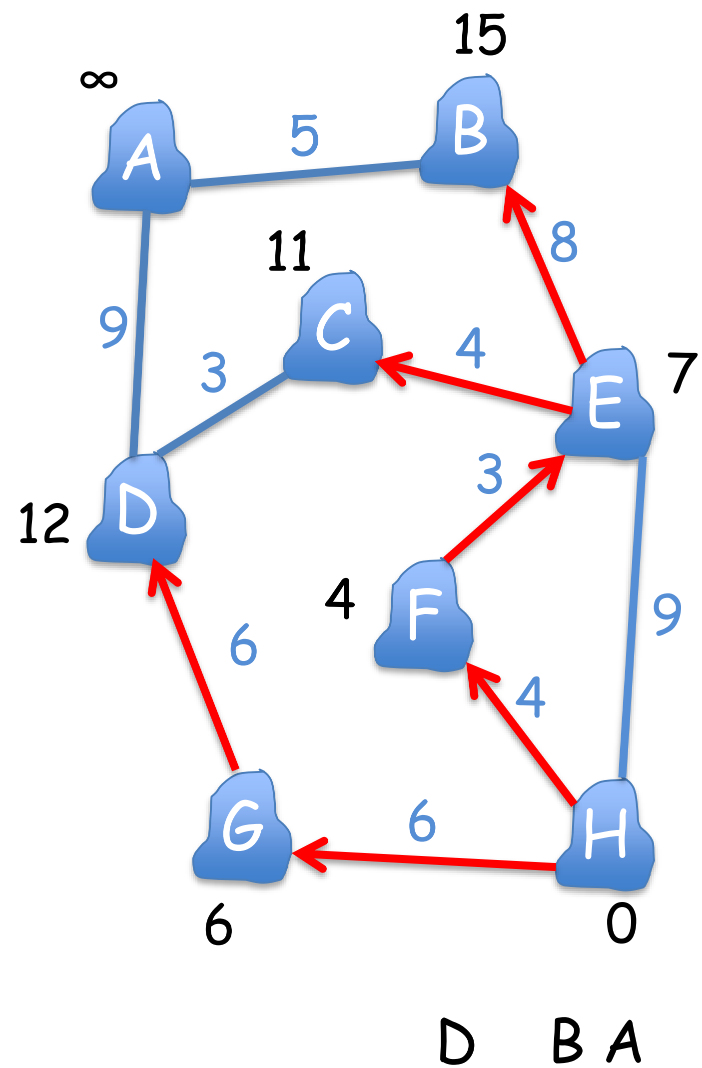


S = {H,F,G,E}

Dijkstra's Algorithm

Dijkstra(G,s)

for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
        update Q

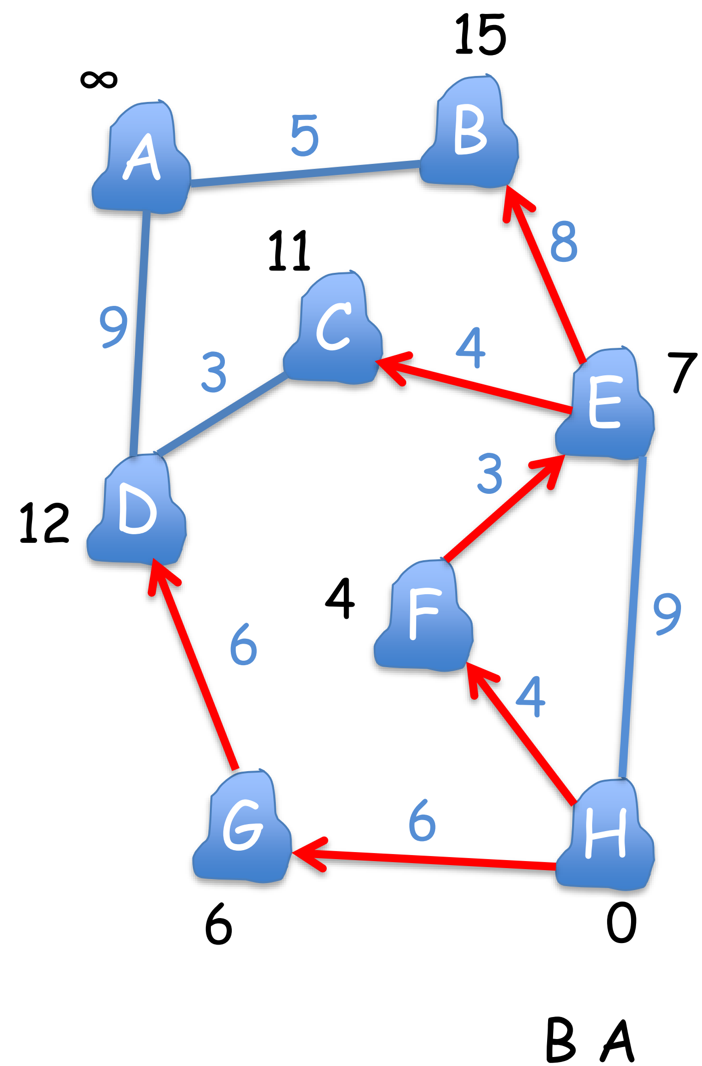


S = {H,F,G,E,C}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
           update Q
```

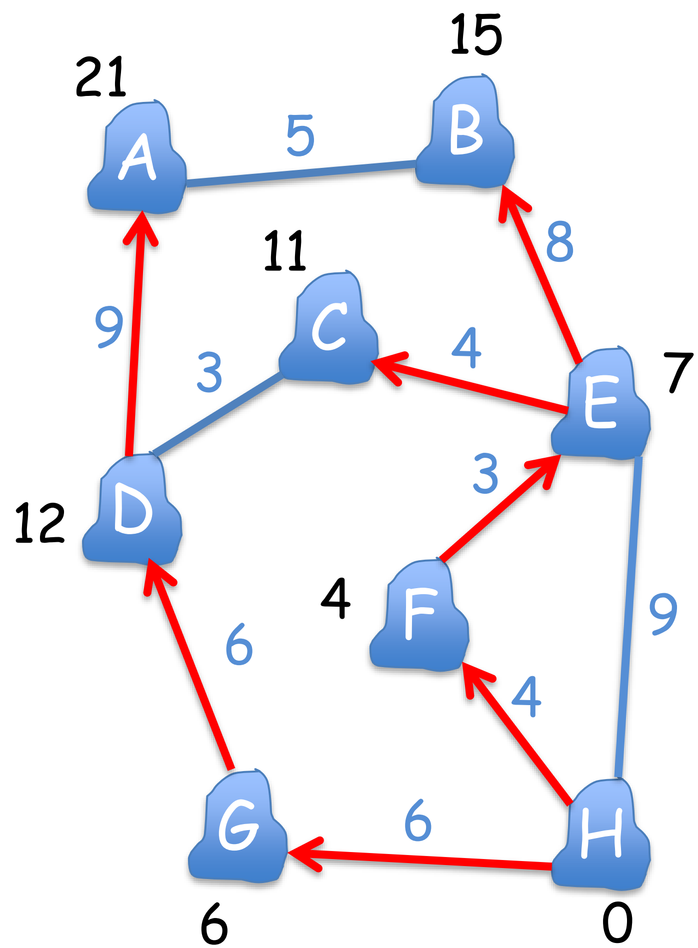


S = {H,F,G,E,C}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                 v.dis = u.dis + w(u,v)
                 v.par = u
           update Q
```



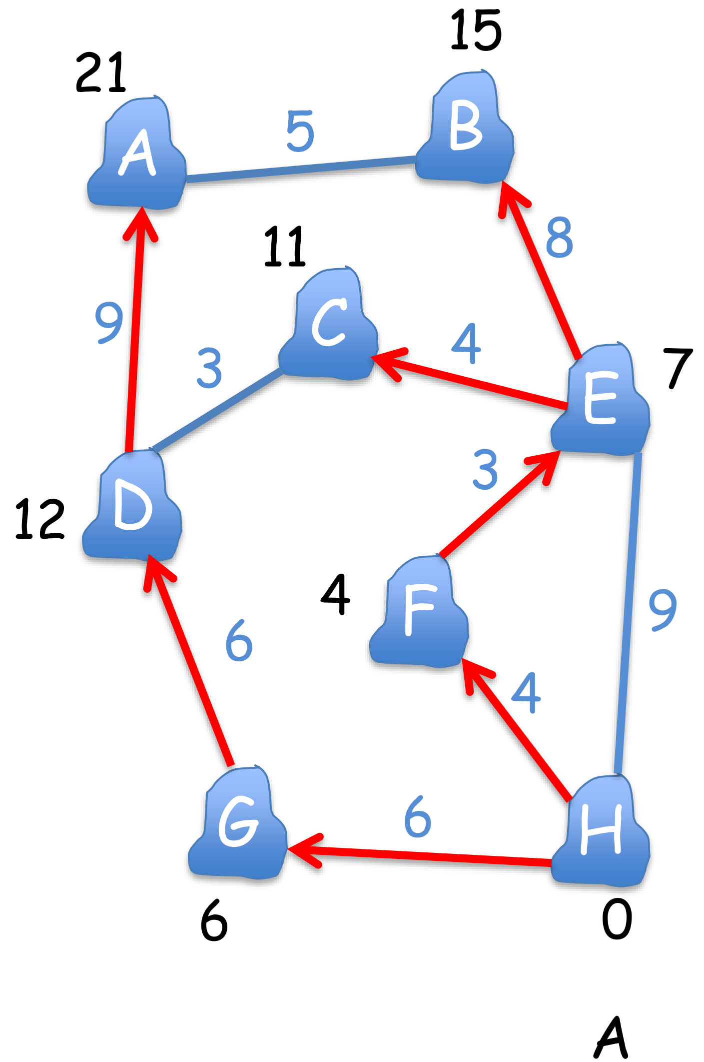
B A

S = {H,F,G,E,C,D}

Dijkstra's Algorithm

Dijkstra(G,s)

for each u of V
    u.key = ∞
    u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v of Adj(u)
        if v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
        update Q

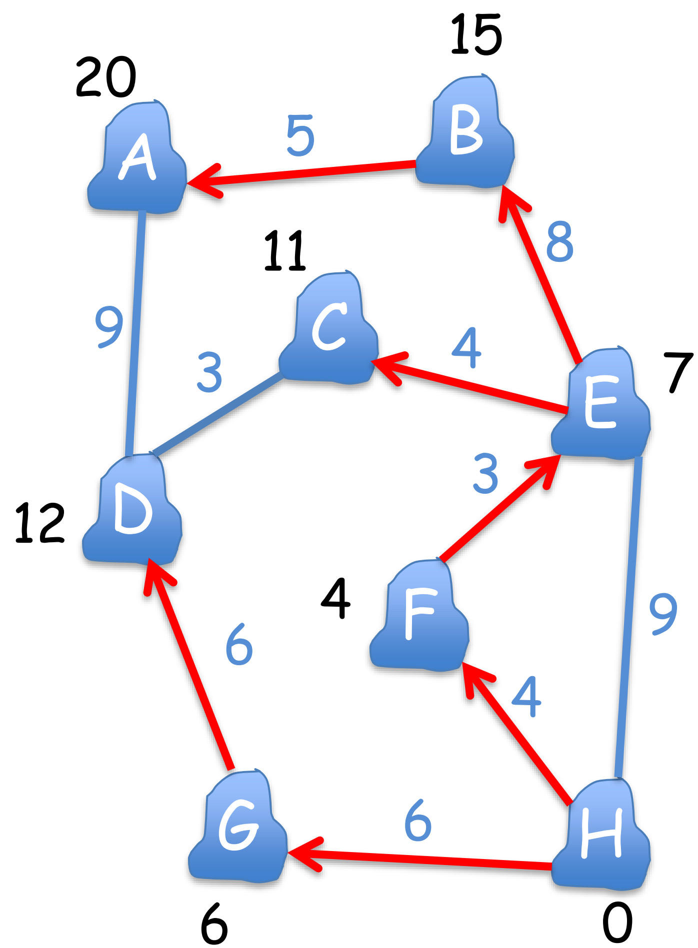


S = {H,F,G,E,C,D,B}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```



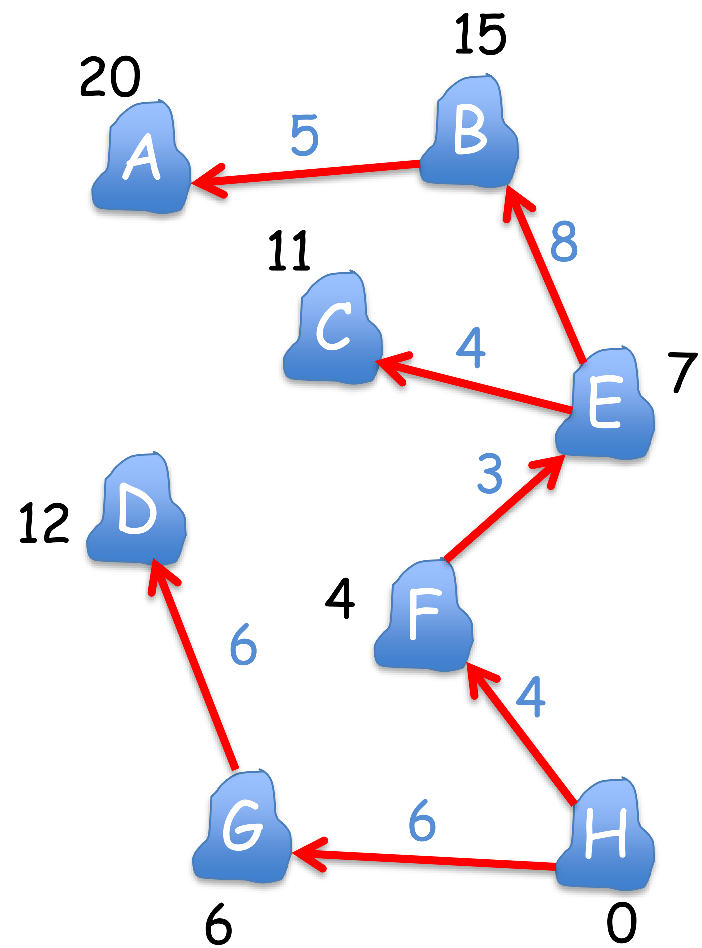
A

S = {H,F,G,E,C,D,B}

Dijkstra's Algorithm

Dijkstra(G,s)

```
for each u of V
      u.key = ∞
      u.par = nil
s.key = 0
initialize an empty set S
create a minimum priority Q on V
while Q ≠ { }
      u = ExtractMin(Q)
      S = S ∪ {u}
      for each v of Adj(u)
            if  v.dis > u.dis + w(u,v)
                  v.dis = u.dis + w(u,v)
                  v.par = u
            update Q
```

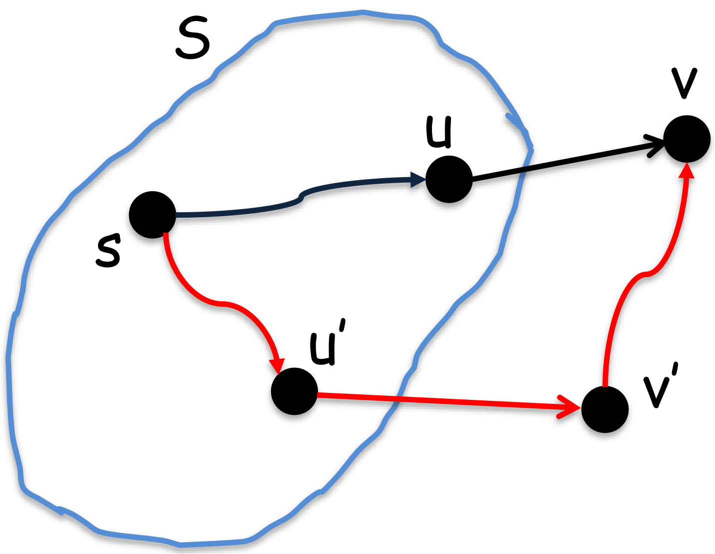


S = {H,F,G,E,C,D,B,A}

Greedy Choice Property

Theorem : For each vertex v in V, v.dis = $\delta(s,v)$ at the time when v is added to S. (Dijkstra's Algorithm computes all shortest path distances correctly)

Proof : Let v be the first vertex that v.dis $\neq \delta(s,v)$ at the time it's added to S. Let's check the true shortest path from s to v.



- v.dis ≤ u.dis + w(u,v)
  = δ(s,u)+w(u,v), since u.dis = δ(s,u)
- v'.dis ≤ u'.dis + w(u',v')
  = δ(s,u')+w(u',v'), since u'.dis = δ(s,u')
- Since v' is in the shortest path from s to v, v'.dis < δ(s,v). From upper bound property, δ(s,v) ≤ v.dis. So v'.dis < v.dis
- Since the vertices extracted from priority queue Q in the order of (…, v, … , v', …), v.dis < v'.dis. Thus, it's a contradiction!