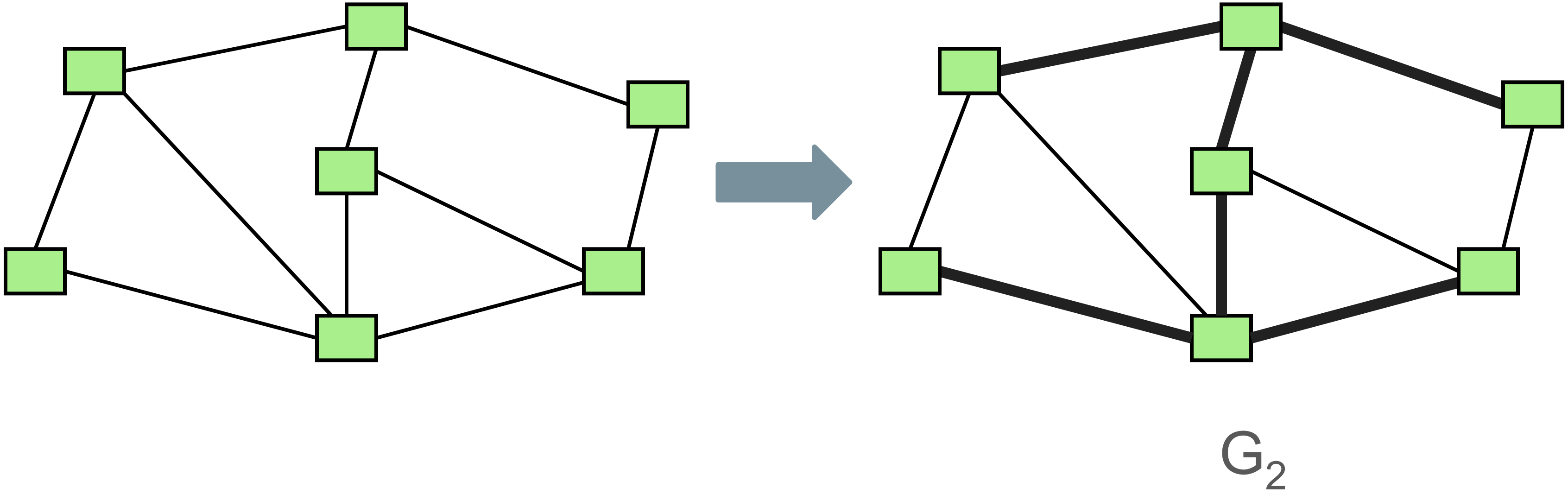
MinimumSpanning Tree

General Problem: Spanning a Graph

A simple problem: Given a *connected* graph G=(V,E), find a minimal subset of the edges such that the graph is still connected

- A graph G_2=(V,E_2) such that G_2 is connected and removing any edge from E_2 makes G_2 disconnected



Observations

1. Any solution to this problem is a tree
    - Recall a tree does not need a root; just means acyclic
    - For any cycle, could remove an edge and still be connected
    - We usually just call the solutions spanning trees

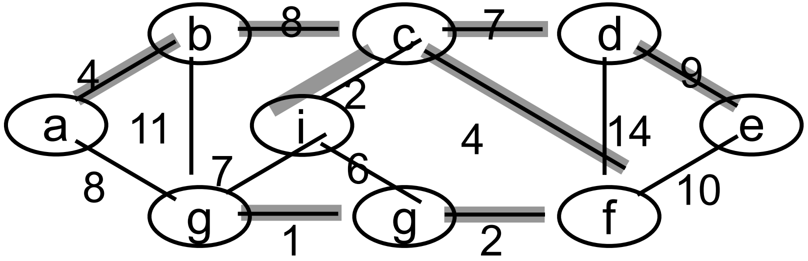
1. Solution not **unique** unless original graph was already a tree

1. Problem ill-defined if original graph not connected
    - We can find a spanning tree per connected component of the graph
    - This is often called a *spanning forest*

1. A tree with **|V|** nodes has **|V|-1** edges
    - This every spanning tree solution has **|V|-1** edges

Minimum Spanning Trees

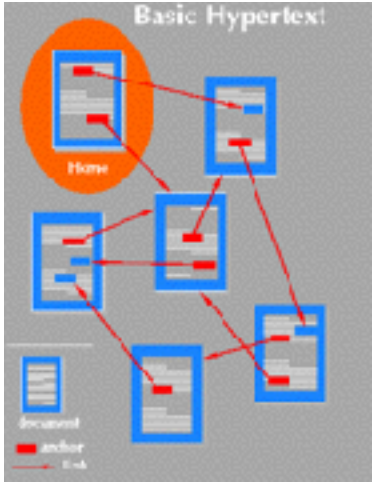
- Spanning Tree
  - A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph
- Minimum Spanning Tree
  - Spanning tree with the **minimum sum of weights**



- Spanning forest
  - If a graph is not connected, then there is a spanning tree for each connected component of the graph

Applications of MST

– Find the least expensive way to connect a set of cities, terminals, computers, etc.



Minimum Spanning Trees

- A connected, undirected graph with:
  - Vertices and Edges
- A **weight** $w(u, v)$ on each edge $(u, v) \in E$

Find $T \subseteq E$ such that:

1. T connects all vertices
2. $w(T) = \Sigma_{(u,v)\in T}\, w(u, v)$ is

   minimized

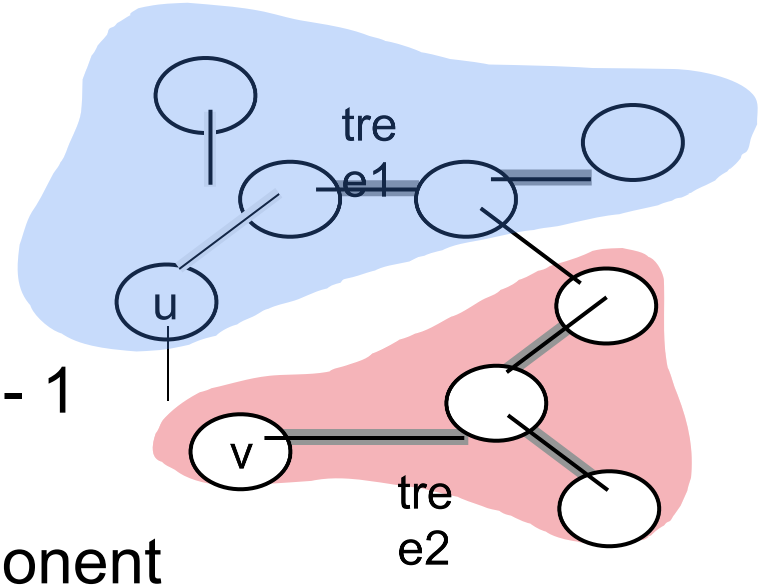
Properties of Minimum Spanning Trees

- Minimum spanning tree is **not** unique

- MST has no cycles –:
  - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- # of edges in a MST: |V| - 1

Kruskal's Algorithm

- Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time.
- Trees are merged together using **safe** edges
- Since an MST has exactly |V| - 1 edges, after |V| - 1 merges, we would have only one component



Central Idea:

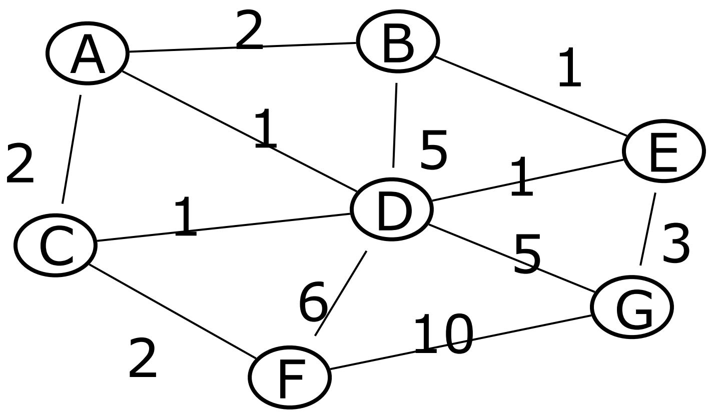
- Grow a forest out of edges that do not grow a cycle, just like for the spanning tree problem.
- But now consider the edges in order by weight

Basic implementation:

- Sort edges by weight → O(|E| `log` |E|) = O(|E| `log` |V|)
- Iterate through edges using for cycle detection
  → O(|E| `log` |V|)

Kruskal's Algorithm

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light** edge that connects them
- Which components to consider at each iteration?
  - Scan the set of edges in monotonically increasing order by weight



Edges in sorted order:

1: (A,D) (C,D) (B,E) (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

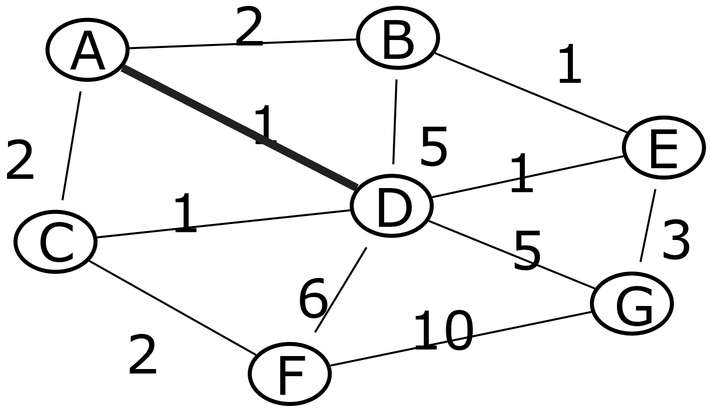
6: (D,F)

10: (F,G)

Sets: (A) (B) (C) (D) (E) (F) (G)

Output:

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ (C,D) (B,E) (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

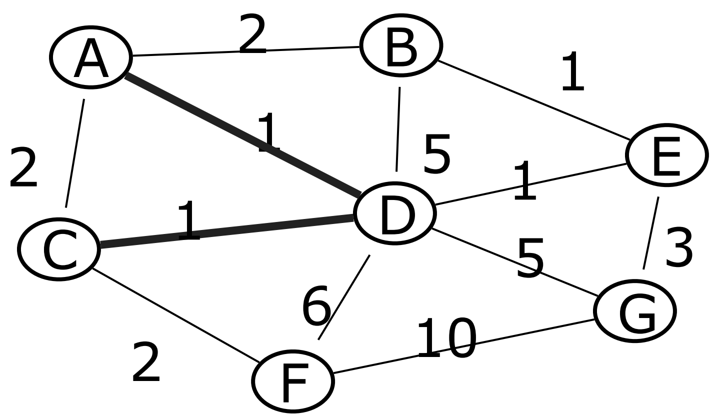
6: (D,F)

10: (F,G)

Sets: (A,D) (B) (C) (E) (F) (G)

Output: (A,D)

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ (B,E) (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

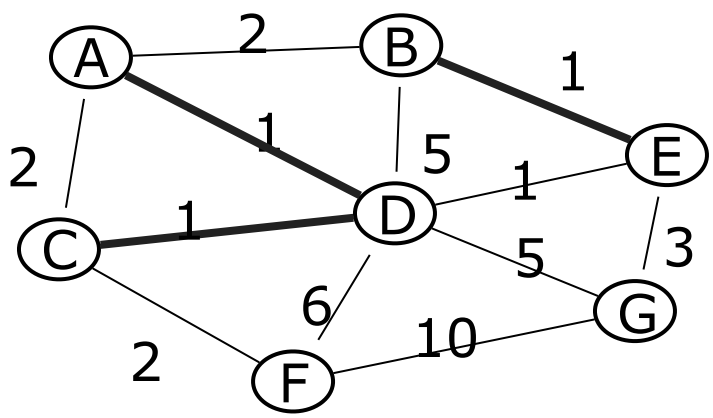
6: (D,F)

10: (F,G)

Sets: (A,C,D) (B) (E) (F) (G)

Output: (A,D) (C,D)

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ (D,E)

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

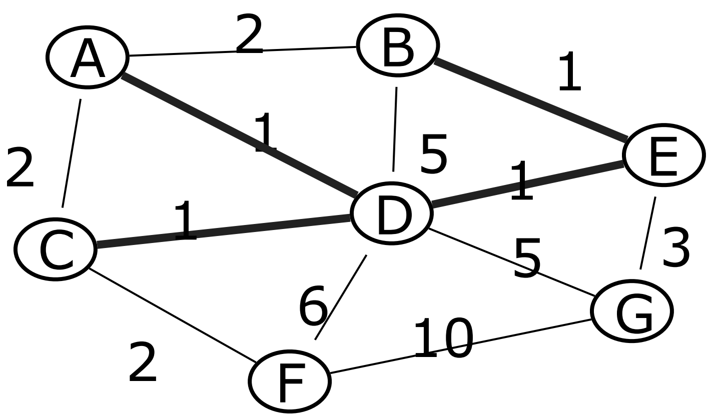
6: (D,F)

10: (F,G)

Sets: (A,C,D) (B,E) (F) (G)

Output: (A,D) (C,D) (B,E)

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: (A,B) (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

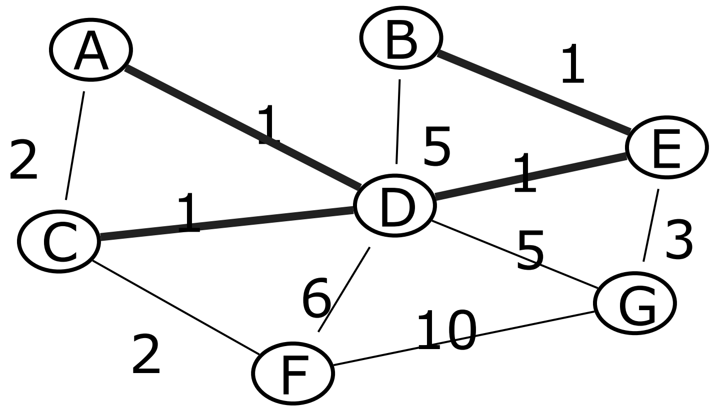
6: (D,F)

10: (F,G)

Sets: (A,B,C,D,E) (F) (G)

Output: (A,D) (C,D) (B,E) (D,E)

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ (C,F) (A,C)

3: (E,G)

5: (D,G) (B,D)

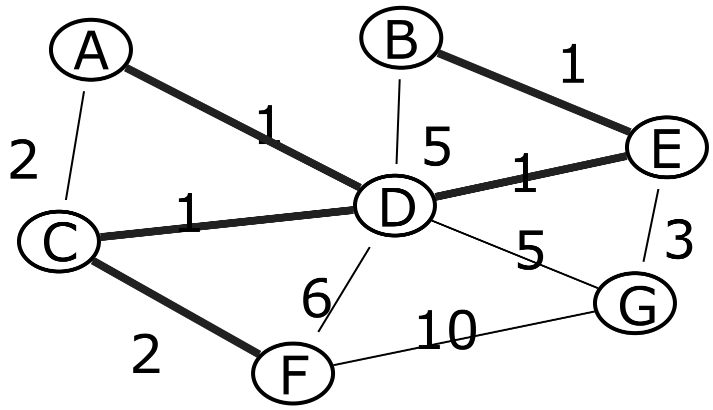
6: (D,F)

10: (F,G)

Sets: (A,B,C,D,E) (F) (G)

Output: (A,D) (C,D) (B,E) (D,E)

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ ~~(C,F)~~ (A,C)

3: (E,G)

5: (D,G) (B,D)

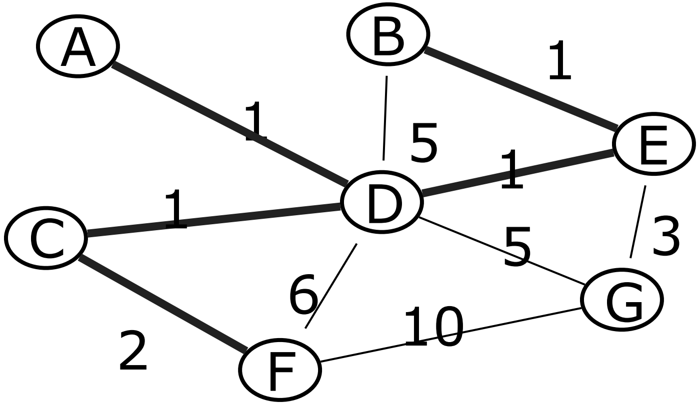
6: (D,F)

10: (F,G)

Sets: (A,B,C,D,E,F) (G)

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

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ ~~(C,F)~~ ~~(A,C)~~

3: (E,G)

5: (D,G) (B,D)

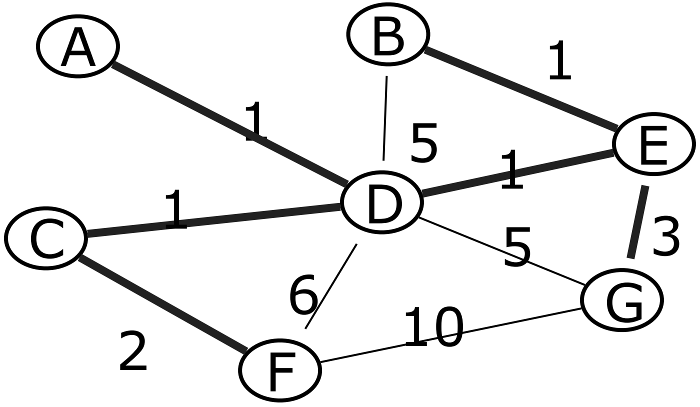
6: (D,F)

10: (F,G)

Sets: (A,B,C,D,E,F) (G)

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

At each step, the union/find sets are the trees in the forest



Edges in sorted order:

1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~

2: ~~(A,B)~~ ~~(C,F)~~ ~~(A,C)~~

3: ~~(E,G)~~

5: (D,G) (B,D)

6: (D,F)

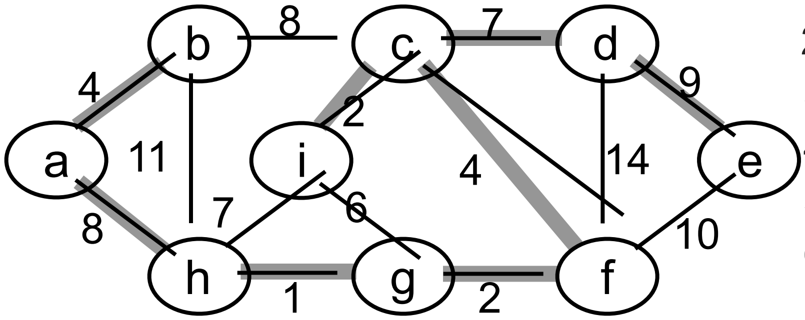
10: (F,G)

Sets: (A,B,C,D,E,F,G)

Output: (A,D) (C,D) (B,E) (D,E) (C,F) (E,G)

At each step, the union/find sets are the trees in the forest

Example



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

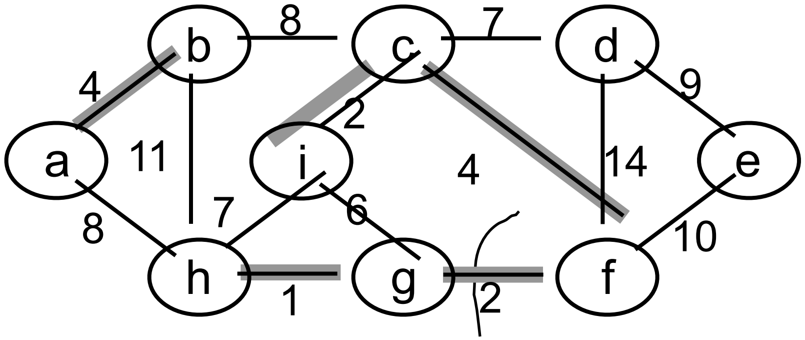
{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

1. Add (h, g)
2. Add (c, i)
3. Add (g, f)
4. Add (a, b)
5. Add (c, f)
6. Ignore (i, g)
7. Add (c, d)
8. Ignore (i, h)
9. Add (a, h)
10. Ignore (b, c)
11. Add (d, e)
12. Ignore (e, f)
13. Ignore (b, h)
14. Ignore (d, f)

{g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
{g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
{g, h, f}, {c, i}, {a}, {b}, {d}, {e}
{g, h, f}, {c, i}, {a, b}, {d}, {e}
{g, h, f, c, i}, {a, b}, {d}, {e}
{g, h, f, c, i}, {a, b}, {d}, {e}
{g, h, f, c, i, d}, {a, b}, {e}
{g, h, f, c, i, d}, {a, b}, {e}
{g, h, f, c, i, d, a, b}, {e}
{g, h, f, c, i, d, a, b}, {e}
{g, h, f, c, i, d, a, b, e}
{g, h, f, c, i, d, a, b, e}
{g, h, f, c, i, d, a, b, e}
{g, h, f, c, i, d, a, b, e}

Implementation of Kruskal's Algorithm

- Uses a disjoint-set data structure to determine <u>whether an edge connects vertices in different components</u>



We would add edge (c, f)

Operations on Disjoint Data Sets

- MAKE-SET(u) – creates a new set whose only member is u
- FIND-SET(u) – returns a representative element from the set that contains u
  - Any of the elements of the set that has a particular property
  - E.g.: $S_u = \{r, s, t, u\}$, the property is that the element be the first one alphabetically

    FIND-SET(u) = r FIND-SET(s) = r
  - FIND-SET has to return the same value for a given set

Operations on Disjoint Data Sets

- UNION(u, v) – unites the dynamic sets that contain u and v, say S_u and S_v
  - E.g.: $S_u = \{r, s, t, u\}$, $S_v = \{v, x, y\}$

    UNION $(u, v) = \{r, s, t, u, v, x, y\}$
- Running time for FIND-SET and UNION depends on implementation.
- Can be shown to be **α(n)=O(lgn)** where **α()** is a very slowly growing function

KRUSKAL(V, E, w)

```
A ← ∅
for each vertex v ∈ V
        do MAKE-SET(v)
sort E into non-decreasing order by w
for each (u, v) taken from the sorted list
      do if FIND-SET(u) ≠ FIND-SET(v)
            then A ← A ∪ {(u, v)}
                  UNION(u, v)
return A
```

O(V)

O(ElgE)

O(E)

O(lgV)

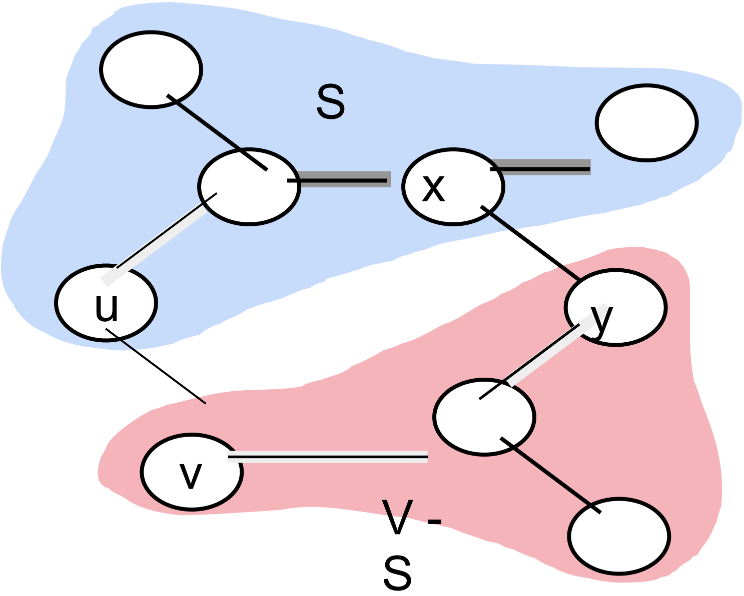
KRUSKAL(V, E, w) (cont.)

```
A ← ∅
for each vertex v ∈ V                                  } O(V)
        do MAKE-SET(v)                                 }
sort E into non-decreasing order by w                  } O(ElgE)
for each (u, v) taken from the sorted list             ← O(E)
     do if FIND-SET(u) ≠ FIND-SET(v)                   }
            then A ← A ∪ {(u, v)}                      } ← O(lgV)
                 UNION(u, v)                           }
return A                                               } O(ElgV)
```

- Running time: $O(V+ElgE+ElgV)=O(ElgE)$
- Since $E=O(V^2)$, we have $lgE=O(2lgV)=O(lgV)$

Kruskal's Algorithm

- Kruskal's algorithm is a **"greedy"** algorithm
- Kruskal's greedy strategy produces a globally optimum solution



Analysis: Kruskal's Algorithm

Correctness: It is a spanning tree

- When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
- The graph is connected, we consider all edges

Correctness: That it is minimum weight

- Can be shown by induction
- At every step, the output is a subset of a minimum tree

Run-time

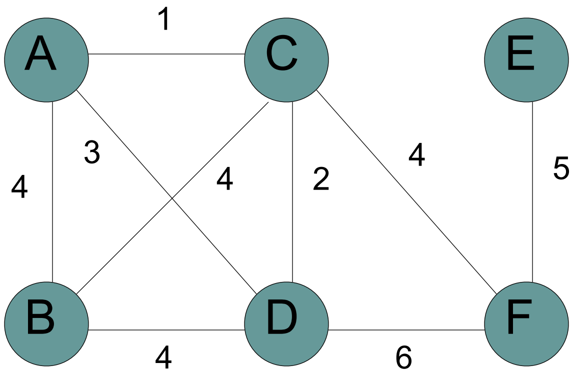
- $O(|\mathbf{E}| \log |\mathbf{V}|)$

Prim's algorithm

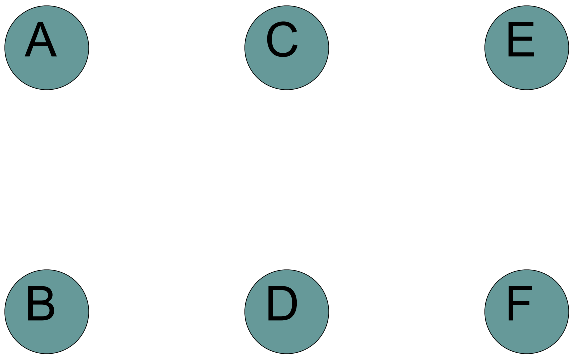
```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

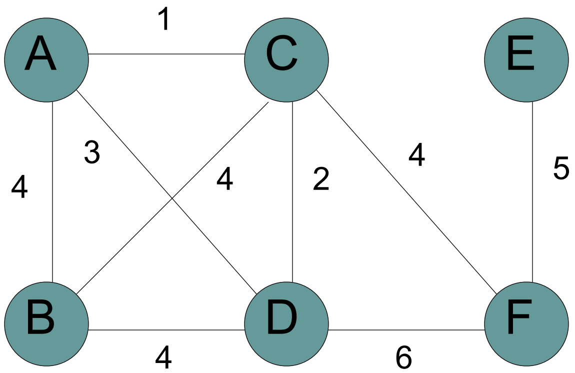


MST

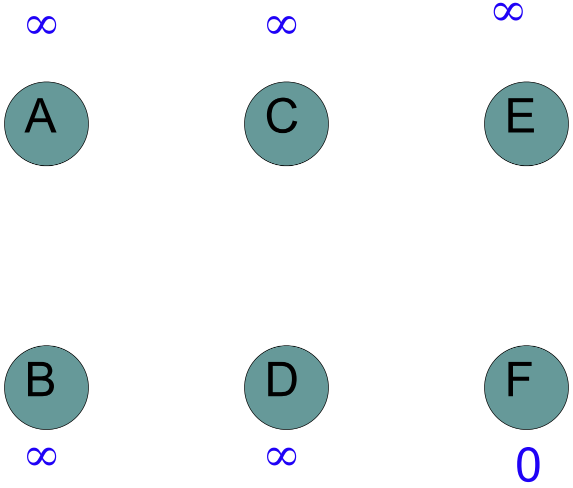


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

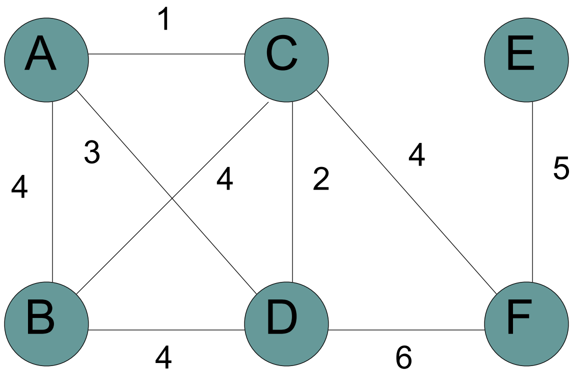


MST

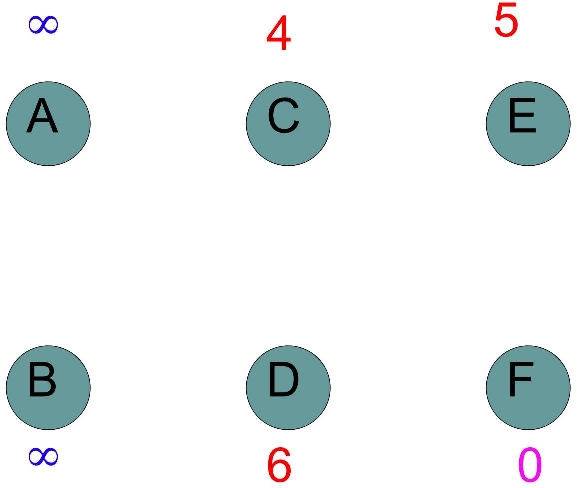


```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

Prim's

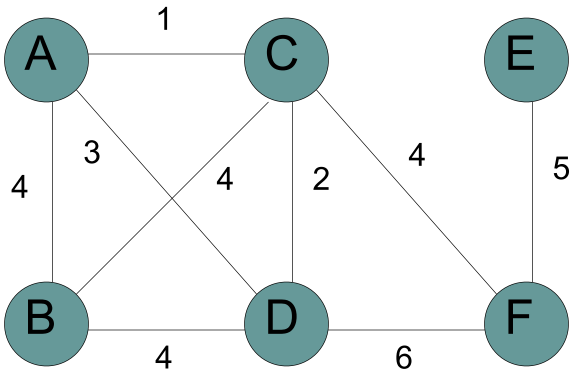


MST

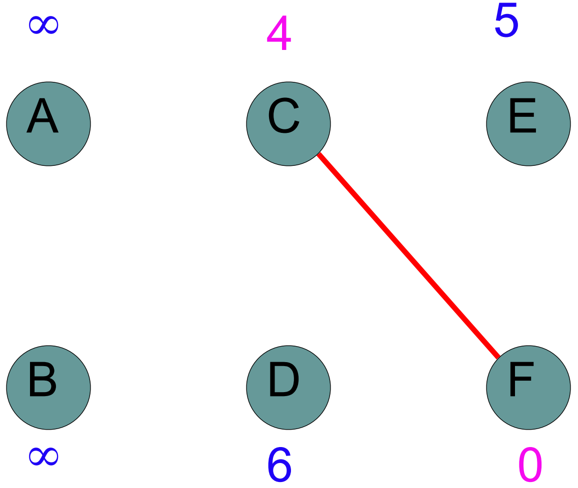


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u,v) ∈ E
        if !visited[v] and w(u,v) < key(v)
            Decrease-Key(v, w(u,v))
            prev[v] ← u
```

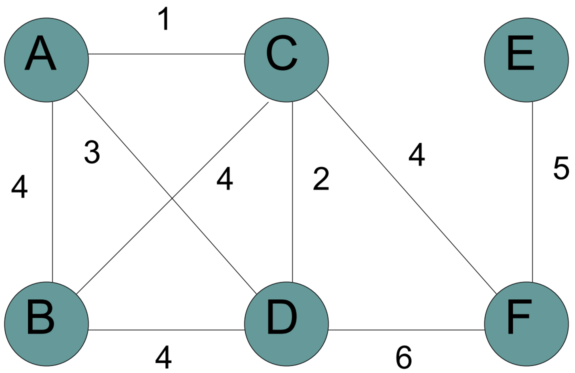


MST

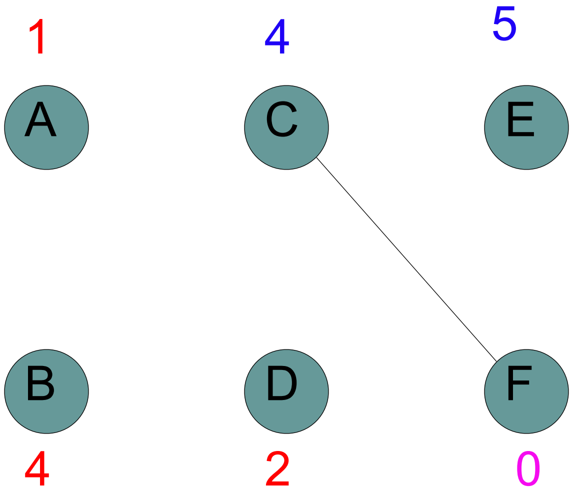


Prim's

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
            if !visited[v] and w(u, v) < key(v)
                Decrease-Key(v, w(u, v))
                prev[v] ← u
```

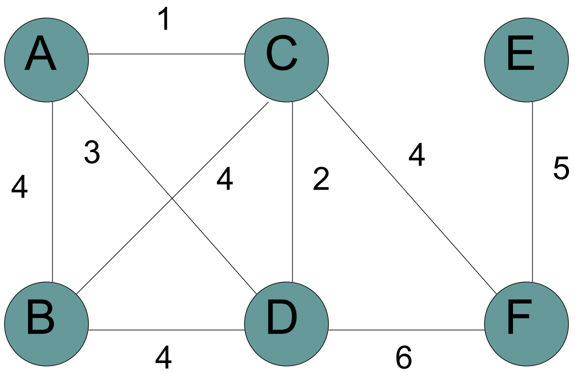


MST

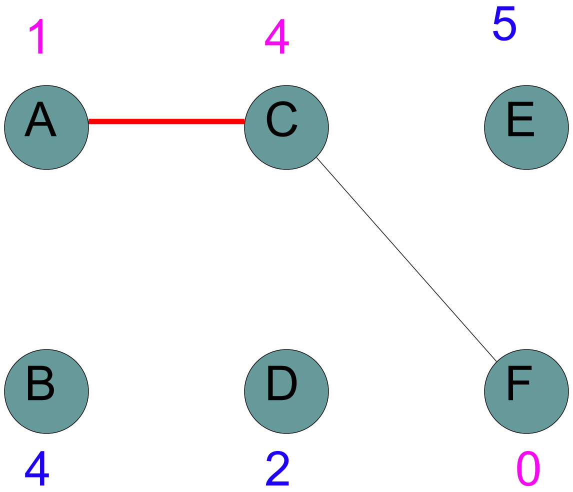


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

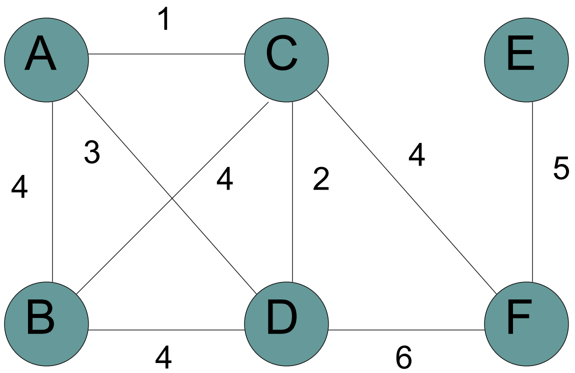


MST

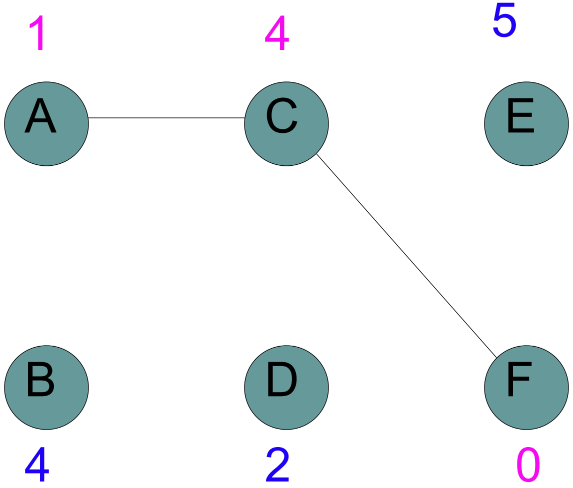


Prim's

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

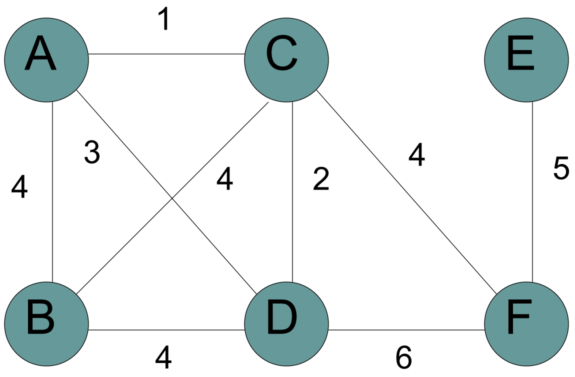


MST

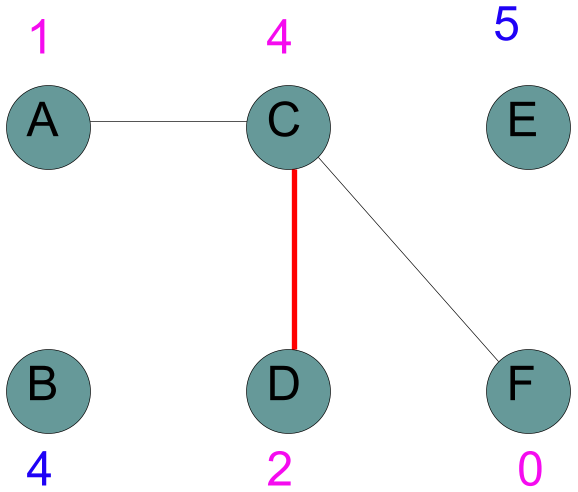


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

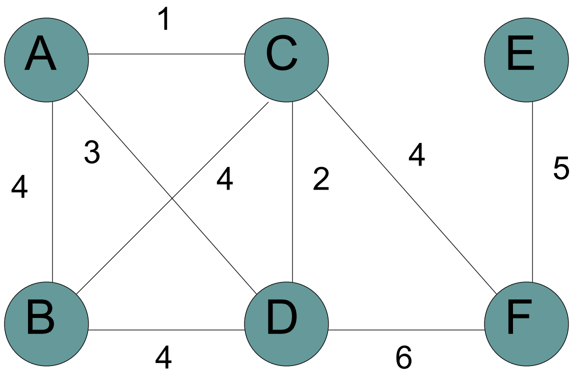


MST

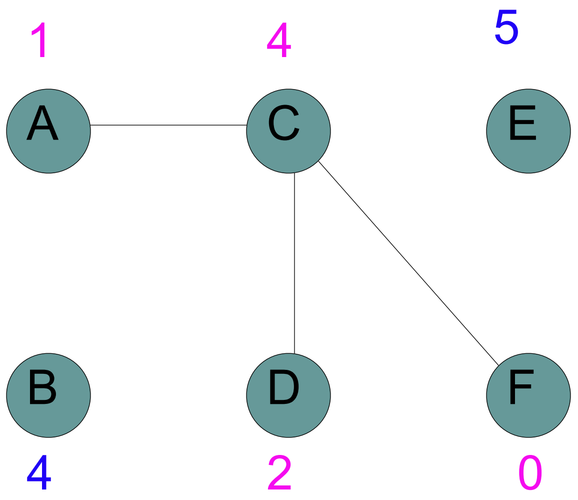


Prim's

```
while !Empty(H)
    u ← EXTRACT-MIN(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            DECREASE-KEY(v, w(u, v))
            prev[v] ← u
```

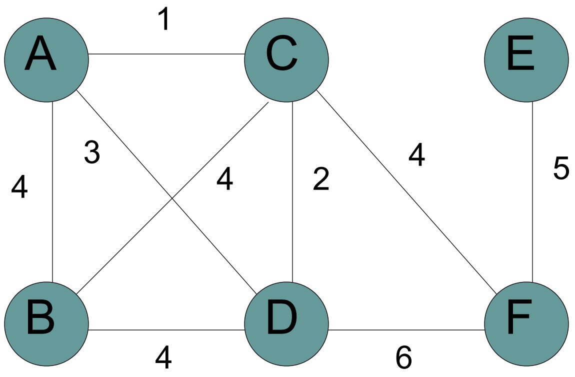


MST

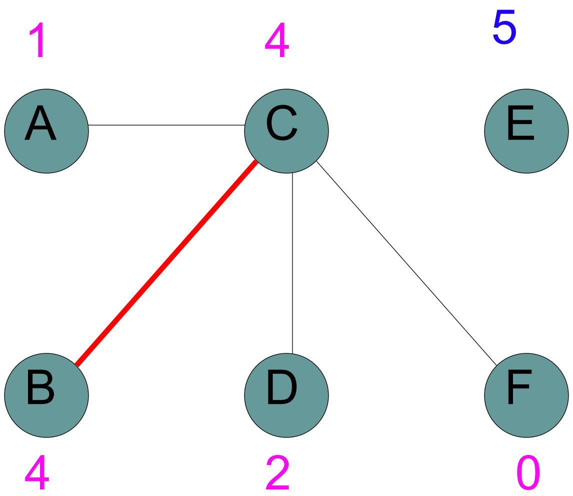


Prim's

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
            if !visited[v] and w(u, v) < key(v)
                Decrease-Key(v, w(u, v))
                prev[v] ← u
```

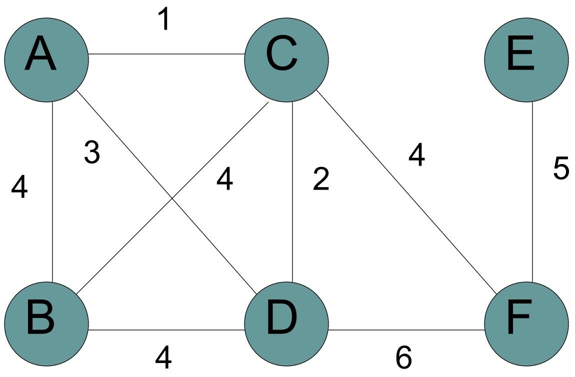


MST

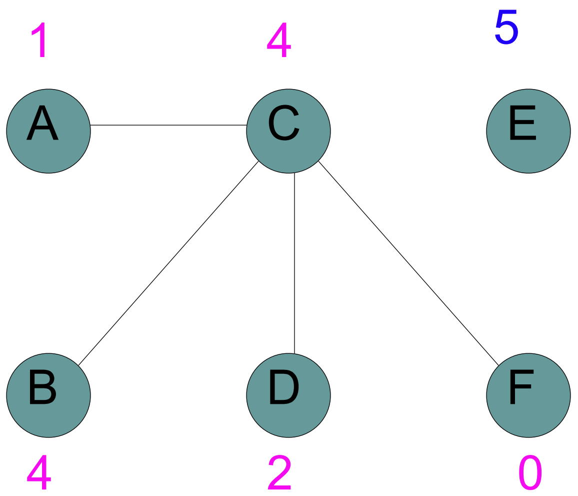


```
while !Empty(H)
    u ← EXTRACT-MIN(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            DECREASE-KEY(v, w(u, v))
            prev[v] ← u
```

Prim's

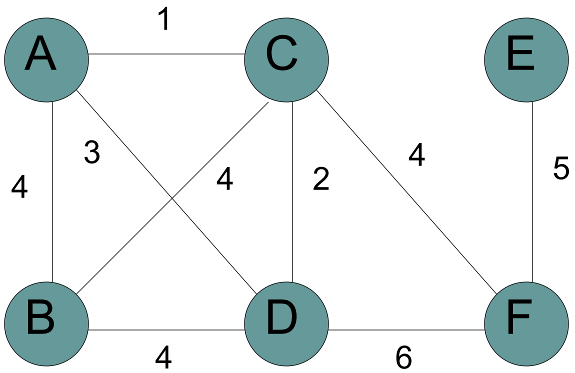


MST

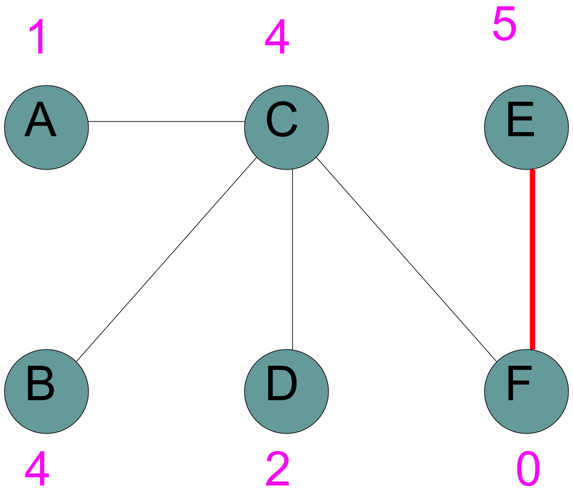


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```



MST



Running time of Prim's

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

Θ(|V|)

Running time of Prim's

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

$\Theta(|V|)$

Running time of Prim's

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

|V| calls to Extract-Min

Running time of Prim's

```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

|E| calls to Decrease-Key

Running time of Prim's

- Same as Dijksta's algorithm

	1 MakeHeap	$\lvert V \rvert$ ExtractMin	$\lvert E \rvert$ DecreaseKey	Total
Array	$O(\lvert V \rvert)$	$O(\lvert V \rvert^2)$	$O(\lvert E \rvert)$	$O(\lvert V \rvert^2)$
Bin heap	$O(\lvert V \rvert)$	$O(\lvert V \rvert \log \lvert V \rvert)$	$O(\lvert E \rvert \log \lvert V \rvert)$	$O((\lvert V \rvert+\lvert E \rvert) \log \lvert V \rvert)$ $O(\lvert E \rvert \log \lvert V \rvert)$
Fib heap	$O(\lvert V \rvert)$	$O(\lvert V \rvert \log \lvert V \rvert)$	$O(\lvert E \rvert)$	**$O(\lvert V \rvert \log \lvert V \rvert + \lvert E \rvert)$**

Kruskal's: $O(\lvert E \rvert \log \lvert E \rvert)$