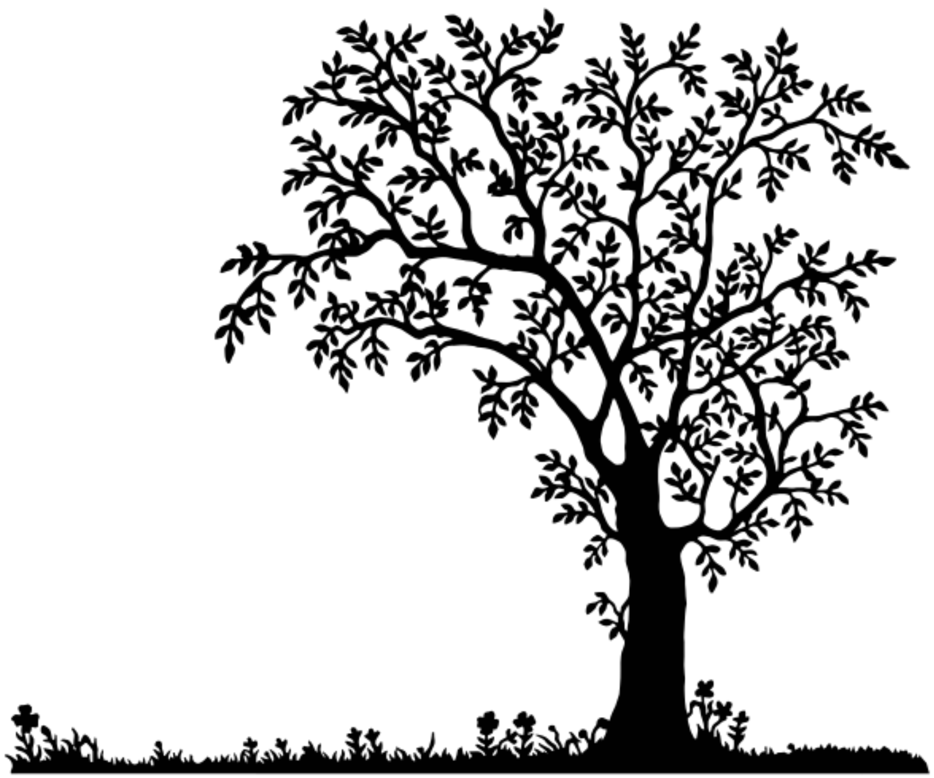
Data Structures and Algorithms (DSA) for AI

Fernanda Madeiral



Minimum spanning tree algorithms

Prim's algorithm

Kruskal's algorithm

Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G
```

Prim’s algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X
```

Prim’s algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}            // s is a randomly chosen vertex
T := ∅              // the edges in T that span X

// main loop
while X ≠ V do
```

Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}            // s is a randomly chosen vertex
T := ∅              // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
```

Prim's algorithm

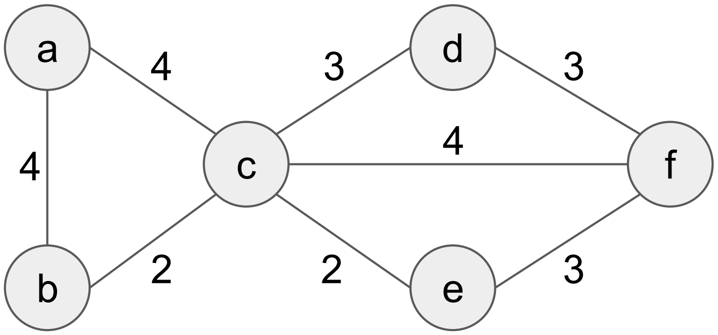
```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
```

Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}            // s is a randomly chosen vertex
T := ∅              // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

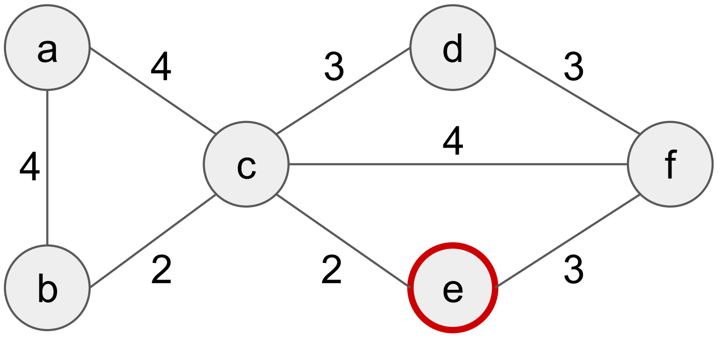
Prim's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```



Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}            // s is a randomly chosen vertex
T := ∅              // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

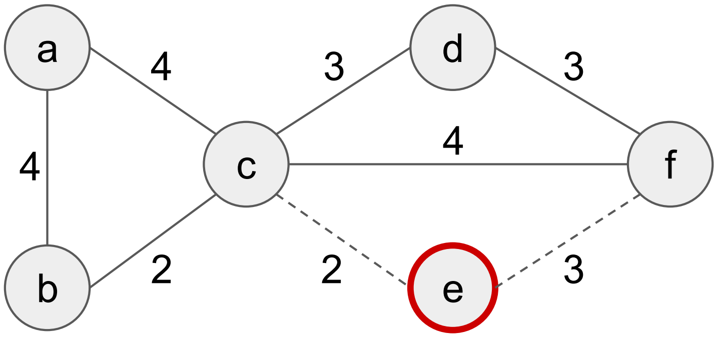


Initialization:

X := {e}
T := ∅

Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}              // s is a randomly chosen vertex
T := ∅                // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

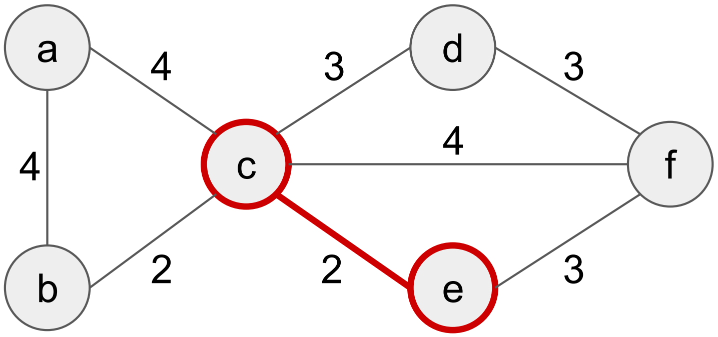


1st loop iteration:

Which edge will be chosen?

Prim’s algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

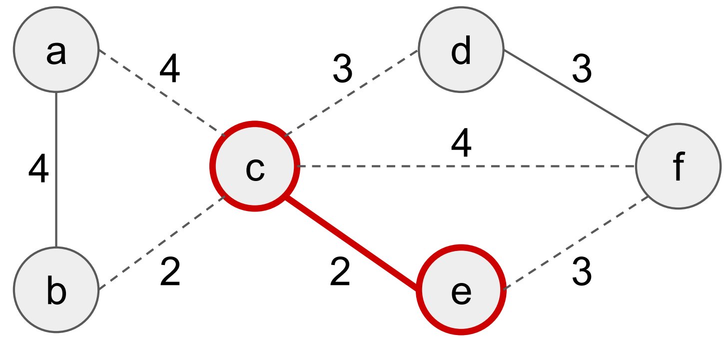


1st loop iteration:

X := {e, c}
T := {(e, c)}

Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}            // s is a randomly chosen vertex
T := ∅              // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

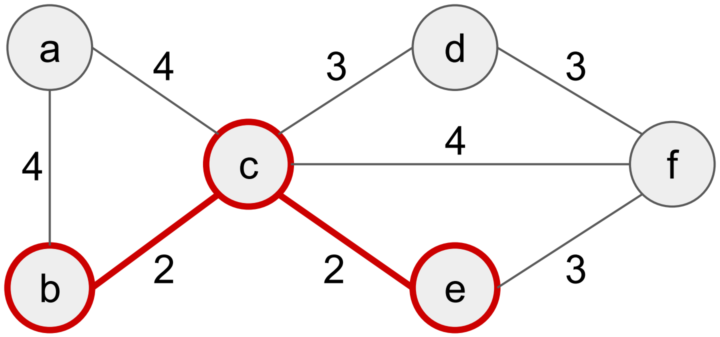


2nd loop iteration:

Which edge will be chosen?

Prim's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

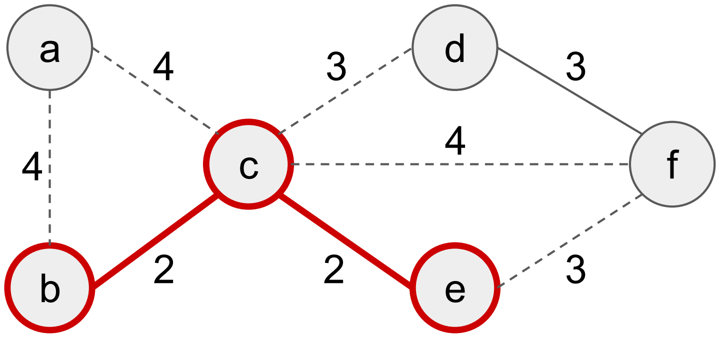


2nd loop iteration:

X := {e, c, b}
T := {(e, c), (c, b)}

Prim's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

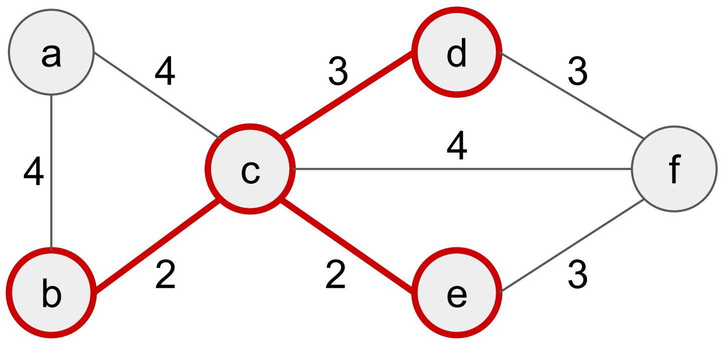


3rd loop iteration:

Which edge will be chosen?

Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}              // s is a randomly chosen vertex
T := ∅                // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

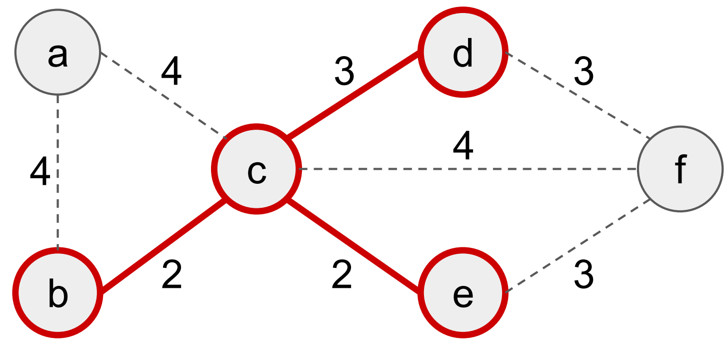


3rd loop iteration:

X := {e, c, b, d}
T := {(e, c), (c, b), (c, d)}

Prim's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

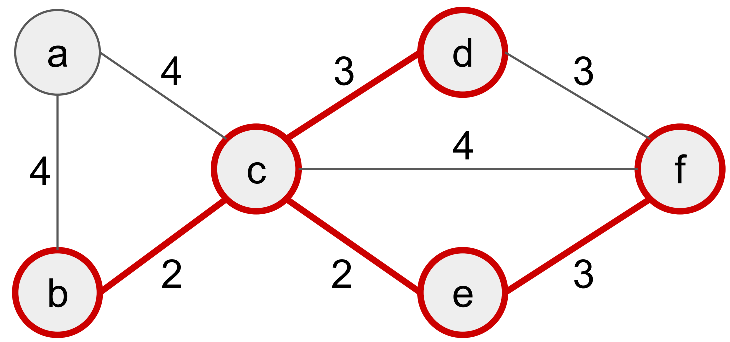


4th loop iteration:

Which edge will be chosen?

Prim’s algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

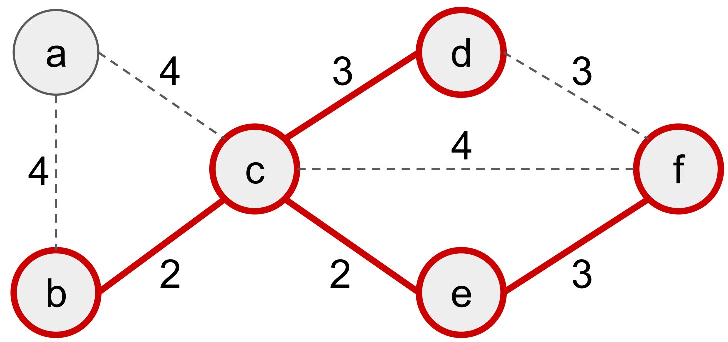


4th loop iteration:

X := {e, c, b, d, f}
T := {(e, c), (c, b), (c, d), (e, f)}

Prim's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

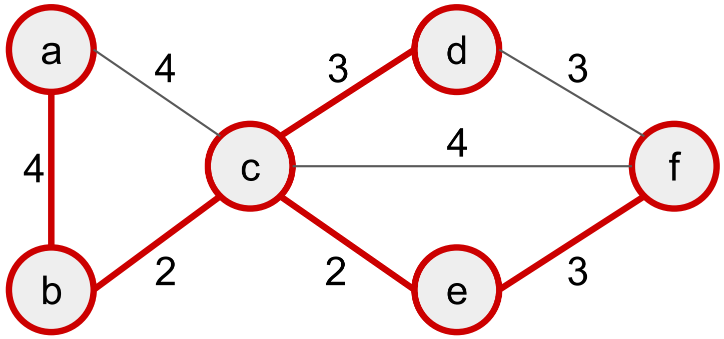


5th loop iteration:

Which edge will be chosen?

Prim’s algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}               // s is a randomly chosen vertex
T := ∅                 // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

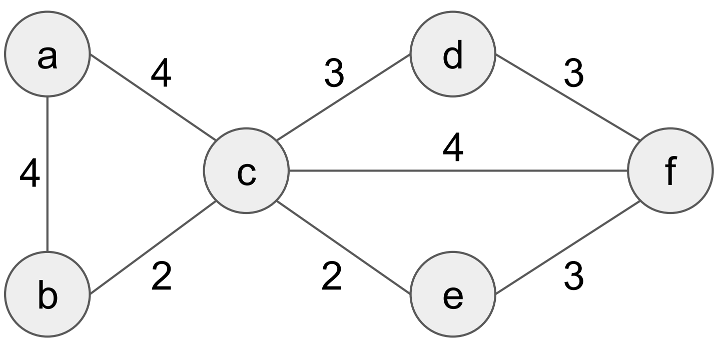


5th loop iteration:

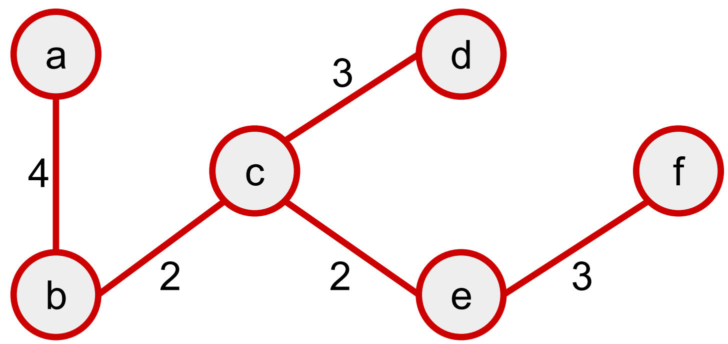
X := {e, c, b, d, f, a}

T := {(e, c), (c, b), (c, d), (e, f), (b, a)}

Prim’s algorithm

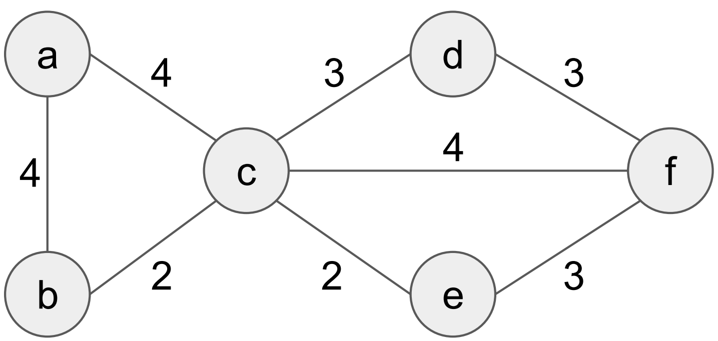


G

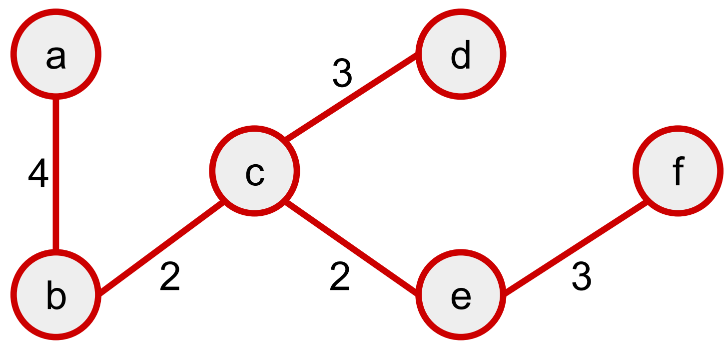


The minimum
spanning tree T
has cost = ?

Prim’s algorithm



G



The minimum spanning tree T has cost = **14**

Prim’s algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

Complexity ("naive" implementation):

?

Prim's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}             // s is a randomly chosen vertex
T := ∅               // the edges in T that span X

// main loop
while X ≠ V do
    find (v, w) that has the least cost edge such that
        v ∈ X and w ∈ V-X
    X := X ∪ {w}
    T := T ∪ {(v, w)}
return T
```

Complexity ("naive" implementation):

Iterations: O(|V|) = O(n)
Edge search: O(|E|) = O(m)

O(n*m)

Prim's algorithm with heap

The objects in the heap are the vertices not-yet-processed, i.e., `V-X`

The key of a vertex `w` $\in$ `V-X` is the minimum cost of an edge `(v, w)` with `v` $\in$ `X`, or $+\infty$ if no such edge exists

Operations we need: Insert, Delete, and ExtractMin, all O(logn)

Prim's algorithm with heap

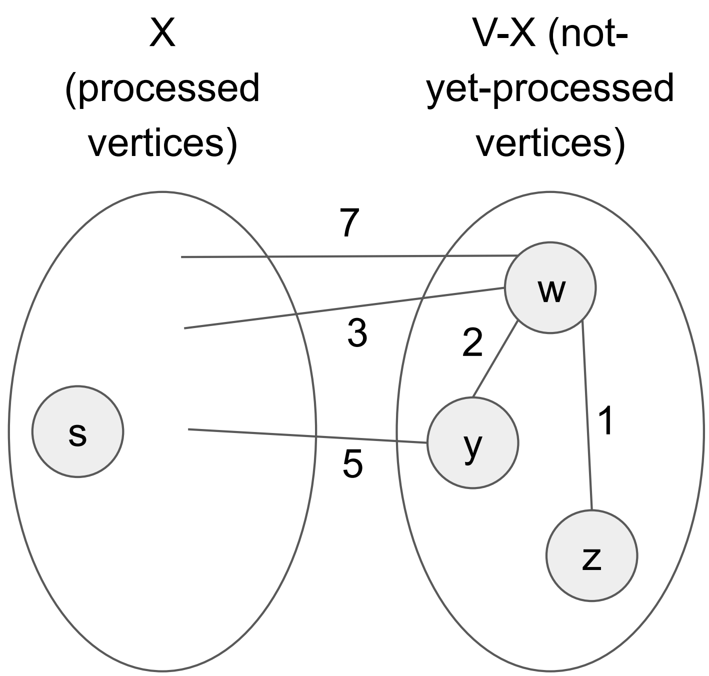
```
Input: G = (V, E)
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}
T := ∅
H := empty heap

for every v ∈ V, v ≠ s do
    if there is an edge (s, v) ∈ E then
        key(v) := c_sv
        cheapestEdge(v) := (s, v)
    else
        key(v) := +∞
        cheapestEdge(v) := null
    Insert v into H
```

Prim's algorithm with heap

```
Input: G = (V, E)
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}
T := ∅
H := empty heap

for every v ∈ V, v ≠ s do
    if there is an edge (s, v) ∈ E then
        key(v) := c_sv
        cheapestEdge(v) := (s, v)
    else
        key(v) := +∞
        cheapestEdge(v) := null
    Insert v into H
```

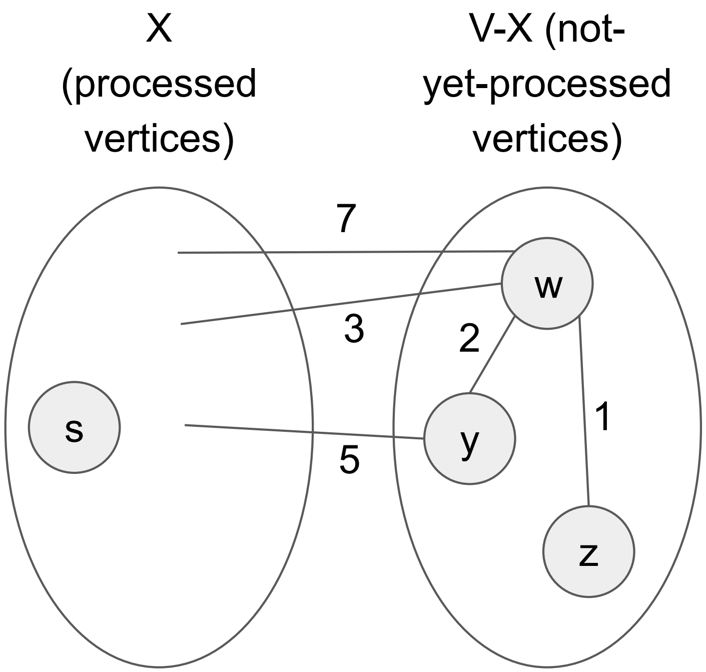
```
(cont)

// main loop
while H is non-empty do
    w := ExtractMin(H)
    X := X ∪ {w}
    T := T ∪ {cheapestEdge(w)}

    // TODO update keys

return T
```

Prim’s algorithm with heap



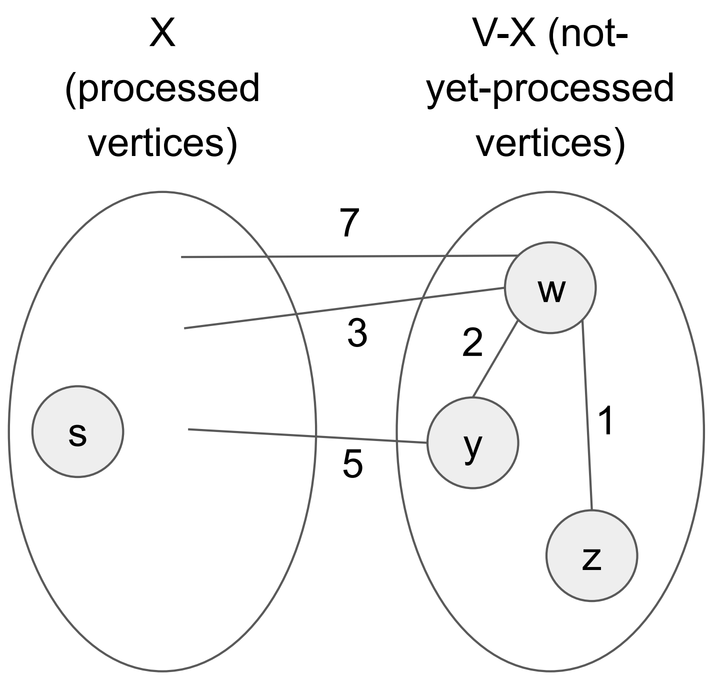
key(w) = ?
key(y) = ?
key(z) = ?

Prim's algorithm with heap



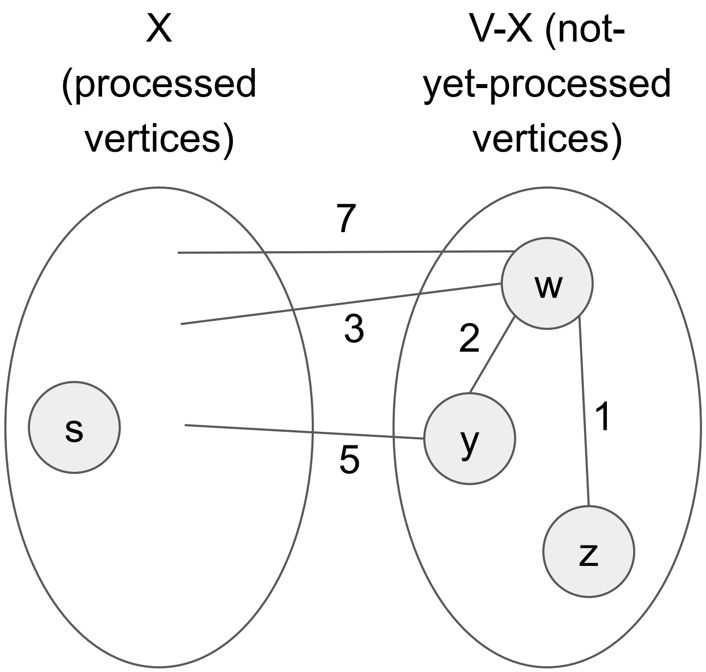
key(w) = 3
key(y) = ?
key(z) = ?

Prim’s algorithm with heap



key(w) = 3
key(y) = 5
key(z) = ?

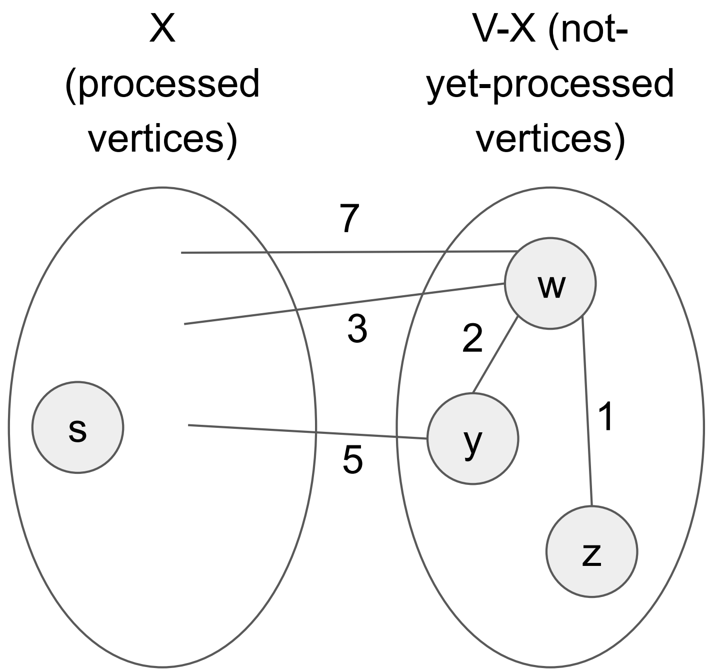
Prim's algorithm with heap



key(w) = 3
key(y) = 5
key(z) = +∞

Prim's algorithm with heap

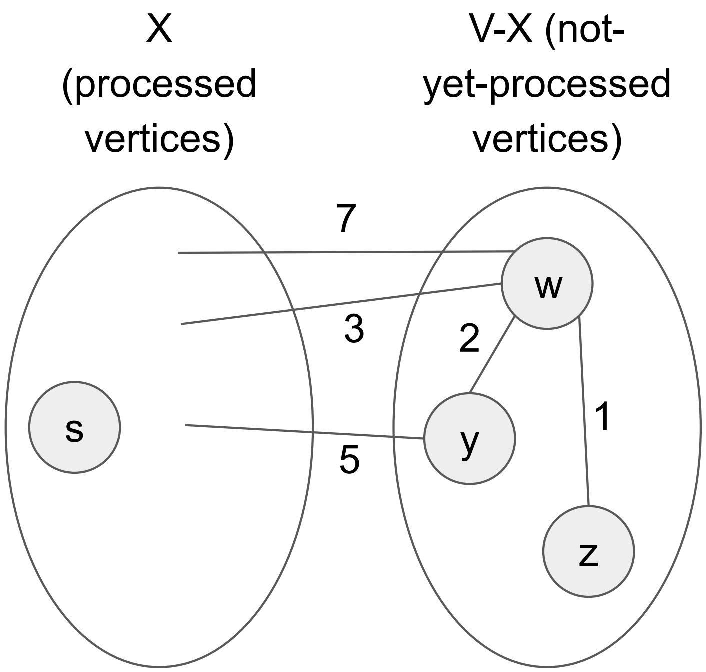
Suppose the vertex w moves to X. What should be the new values of key(y) and key(z)?



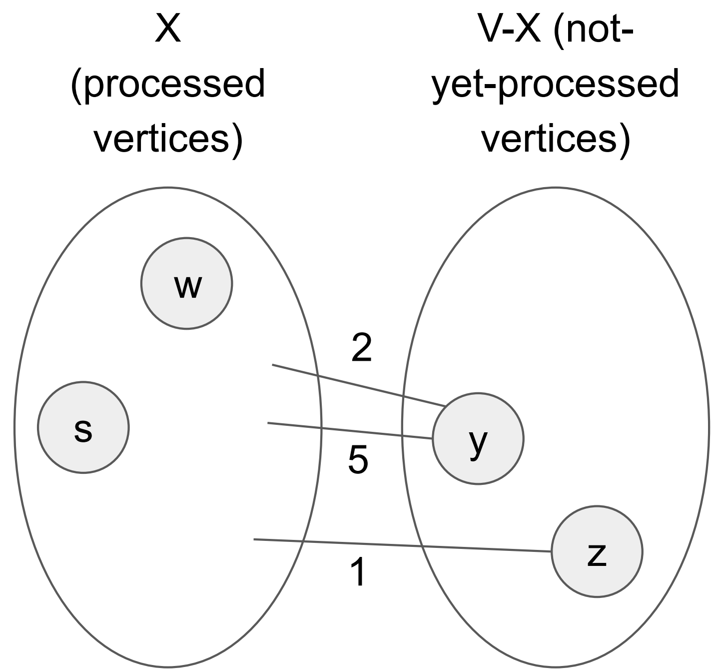
key(w) = 3
key(y) = 5
key(z) = +∞

Prim's algorithm with heap

Suppose the vertex w moves to X. What should be the new values of key(y) and key(z)?



key(w) = 3
key(y) = 5
key(z) = +∞



key(y) = 2
key(z) = 1

Prim's algorithm with heap

```
Input: G = (V, E)
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}
T := ∅
H := empty heap

for every v ∈ V, v ≠ s do
    if there is an edge (s, v) ∈ E then
        key(v) := c_sv
        cheapestEdge(v) := (s, v)
    else
        key(v) := +∞
        cheapestEdge(v) := null
    Insert v into H
```

```
(cont)

// main loop
while H is non-empty do
    w := ExtractMin(H)
    X := X ∪ {w}
    T := T ∪ {cheapestEdge(w)}

    // update keys
    for every edge (w, y) with y ∈ V-X do
        if c_wy < key(y) then
            Delete y from H
            key(y) := c_wy
            cheapestEdge(y) := (w, y)
            Insert y into H

return T
```

Prim’s algorithm with heap

Big-O?

Initialization: O(|V|) * O(log|V|)

Main loop: O(|V|) * O(log|V|) + O(|E|) * O(log|V|)

Total: O((|V|+|E|) * log|V|) = O((n+m) * logn)

```
Input: G = (V, E)
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
X := {s}
T := ∅
H := empty heap

for every v ∈ V, v ≠ s do
    if there is an edge (s, v) ∈ E then
        key(v) := c_sv
        cheapestEdge(v) := (s, v)
    else
        key(v) := +∞
        cheapestEdge(v) := null
    Insert v into H
```

```
(cont)

// main loop
while H is non-empty do
    w := ExtractMin(H)
    X := X ∪ {w}
    T := T ∪ {cheapestEdge(w)}

    // update keys
    for every edge (w, y) with y ∈ V-X do
        if c_wy < key(y) then
            Delete y from H
            key(y) := c_wy
            cheapestEdge(y) := (w, y)
            Insert y into H

return T
```

Minimum spanning tree algorithms

Prim's algorithm

Kruskal's algorithm

Kruskal's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅              // the edges that span G
```

Kruskal's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅              // the edges that span G

// Preprocessing
sort edges of E by cost
```

Kruskal's algorithm

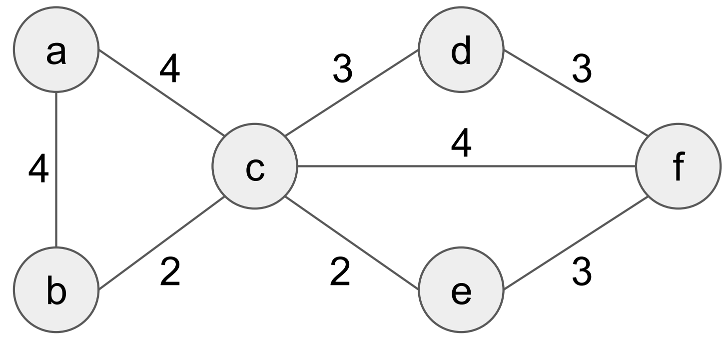
```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅                 // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do

        T := T ∪ {e}
return T
```

Kruskal's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅              // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

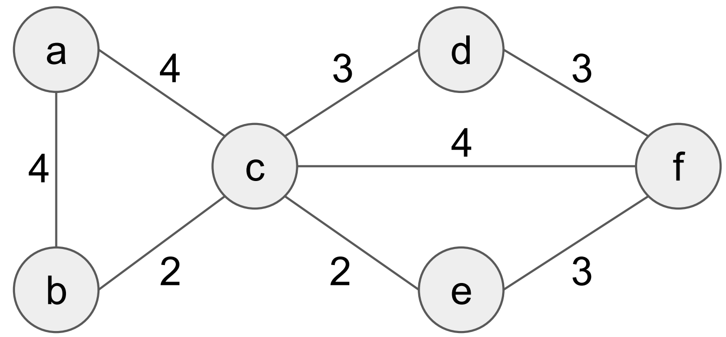
Kruskal's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅               // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```



Kruskal's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅              // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

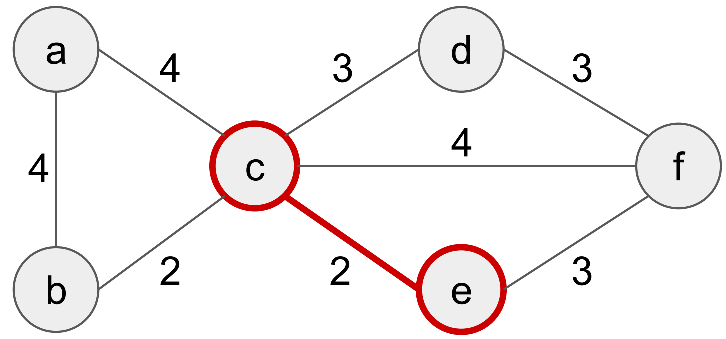


1st loop iteration:

Which edge will be chosen?

Kruskal's algorithm

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅              // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

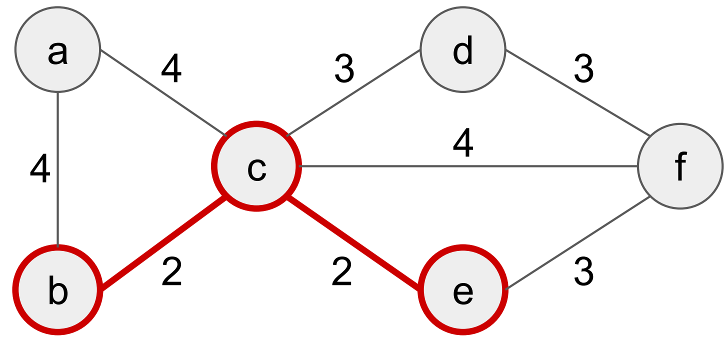


1st loop iteration:

T := {(c, e)}

Kruskal's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅               // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

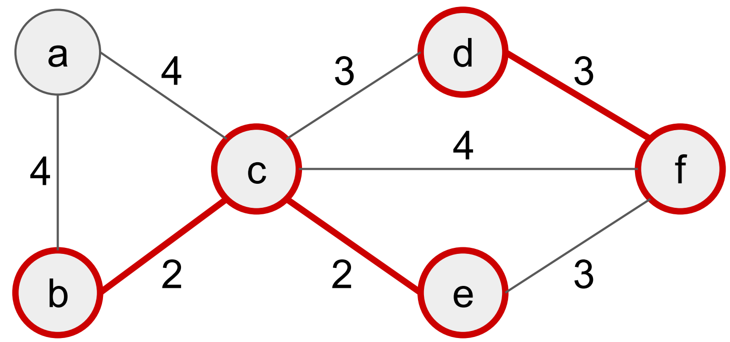


2nd loop iteration:

T := {(c, e), (c, b)}

Kruskal's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅                // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

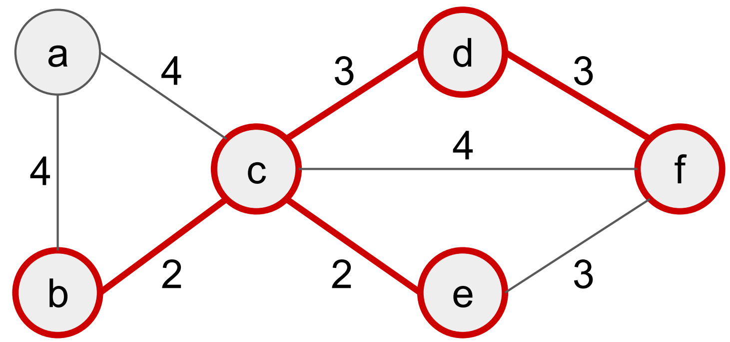


3rd loop iteration:

T := {(c, e), (c, b), (d, f)}

Kruskal's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅               // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

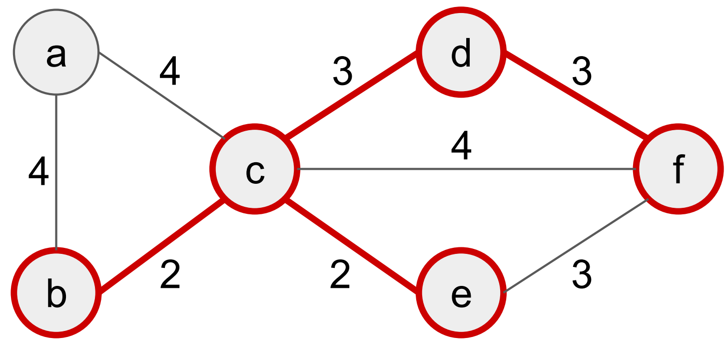


4th loop iteration:

T := {(c, e), (c, b), (d, f), (c, d)}

Kruskal's algorithm

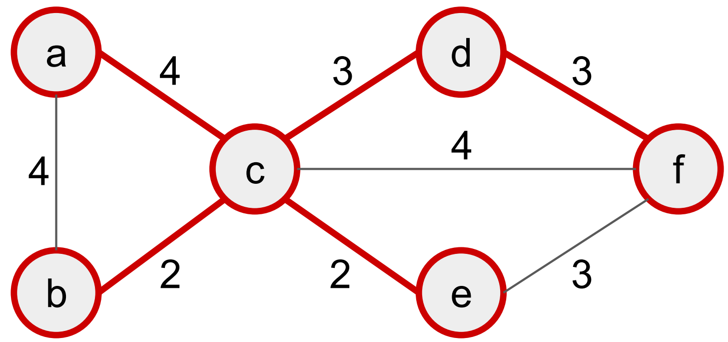
```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅               // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```



Picking (e, f) would create a cycle!

Kruskal’s algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅               // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```



5th loop iteration:

T := {(c, e), (c, b), (d, f), (c, d), (c, a)}

Kruskal's algorithm

```
Input: G = (V, E)    // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅               // the edges that span G

// Preprocessing
sort edges of E by cost

// main loop
for each e ∈ E in increasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

Complexity:

It depends a lot on sorting and acyclic checking!

"Naive" implementation:
Assuming the usage of Merge Sort for sorting and DFS for acyclic checking, that would result in O(m*n).

Kruskal's algorithm - how can we improve it?

How could we check if adding an edge {u, v} in T would create a cycle?
Remember: We create a cycle if u and v are already in the same connected component (or set).

IDEA: we keep track of the so far connected components, so we can quickly check if they belong to the same

- We start with a component for each node.
- Components merge when we add an edge in T.
- Need a way to check if u and v are in same component and to merge two components into one.

Kruskal's algorithm with Union-Find (new data structure!)

The Union-Find abstract data type supports the following operations:

- Initialize(V): given an array V of objects, create a union-find data structure with each object $v \in V$ in its own set.
- Find(U,x): given a union-find data structure and an object x in it, return the name of the set that contains x.
- Union(U,x,y): given a union-find data structure and two objects $x, y \in V$ in it, merge the sets that contain x and y into a single set.

Kruskal’s algorithm with Union-Find (new data structure!)

The Union-Find abstract data type supports the following operations:

- Initialize(V): given an array V of objects, create a union-find data structure with each object $v \in V$ in its own set. **O(n)**

- Find(U,x): given a union-find data structure and an object x in it, return the name of the set that contains x. **O(log n)**

- Union(U,x,y): given a union-find data structure and two objects $x, y \in V$ in it, merge the sets that contain x and y into a single set. **O(log n)**

Kruskal's algorithm with Union-Find

```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅               // the edges that span G
U := Initialize(V)  // union-find data structure
sort edges of E by cost

// main loop
for each (v,w) ∈ E in increasing order of cost do
    if Find(U,v) ≠ Find(U,w) then   // no v-w path in T
        T := T ∪ {(v,w)}
        Union(U,v,w)
return T
```

Kruskal’s algorithm with Union-Find

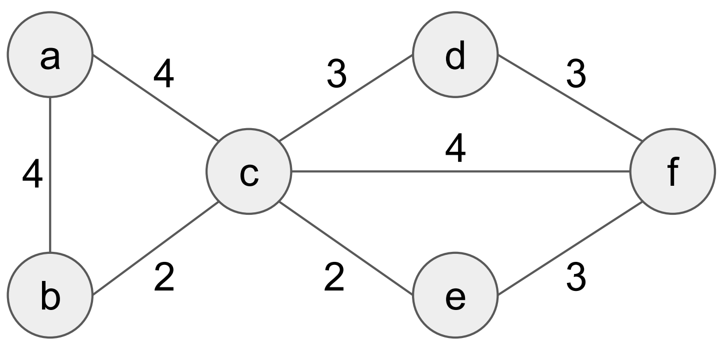
```
Input: G = (V, E)   // G is a connected and undirected graph
       a cost c_e for each edge e ∈ E
Output: the edges of a minimum spanning tree of G

// Initialization
T := ∅                // the edges that span G
U := Initialize(V)   // union-find data structure
sort edges of E by cost

// main loop
for each (v,w) ∈ E in increasing order of cost do
    if Find(U,v) ≠ Find(U,w) then   // no v-w path in T
        T := T ∪ {(v,w)}
        Union(U,v,w)
return T
```

Complexity:

Initialize + sorting: O(n) + O(m log n)
2*m **Find** operations: O(m log n)
n-1 **Union** operations: O(n log n)

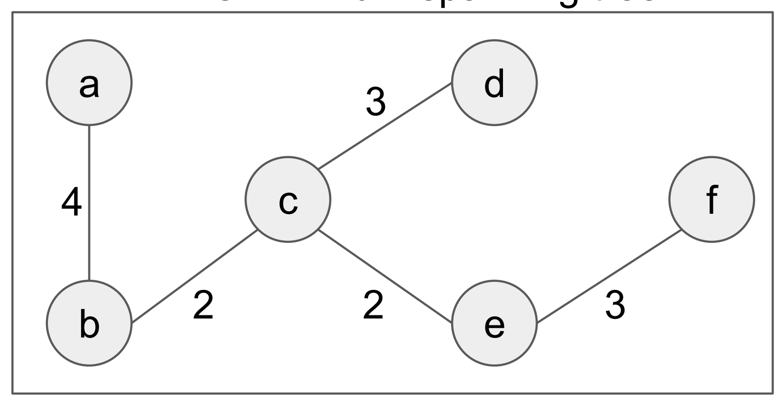
Total: O((m+n) log n)

Prim's and Kruskal's algorithms outputs were…

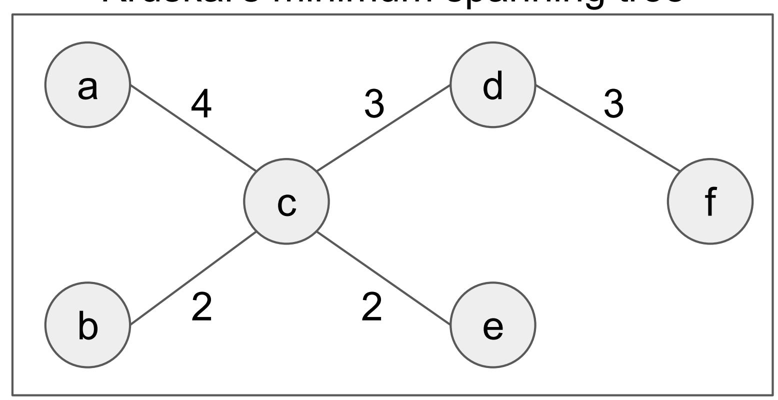


Graph G

Prim's minimum spanning tree



Kruskal's minimum spanning tree



Prim’s and Kruskal’s algorithms

Both are greedy algorithms for minimum spanning tree

Prim grows a tree while Kruskal grows a forest

In which situation one is better than the other one?

Prim's and Kruskal's algorithms

Both are greedy algorithms for minimum spanning tree

Prim grows a tree while Kruskal grows a forest

In which situation one is better than the other one?

> Prim is better for dense graphs while Kruskal is better for sparse graphs

Kruskal Application: single link clustering
Bottom-up clustering (greedy)

```
Input: a set x of data points, a symmetric similarity
function f, and a positive integer k ∈ {1,2,3,...,|X|}

Output: a partition of X into k non-empty clusters


C := ∅ // keeps track of current clusters

for each x ∈ X do

     Add {x} to C


// Main loop

while C contains more than k cluster do

     Remove from C the clusters S_1, S_2 that minimize

     F(S_1,S_2)

     add S_1 ∪ S_2 to C

return C
```

- Important in unsupervised ML
- Goal = partition data into "coherent groups" / clusters
- Similarity function f(x,y) = symmetric, assigns a similarity score to pair data points x,y
- F(S_1,S_2)=min f(x,y), where x∈S_1 and y∈S_2 or "best case" similarity between points in different clusters