香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 20: Minimum spanning tree

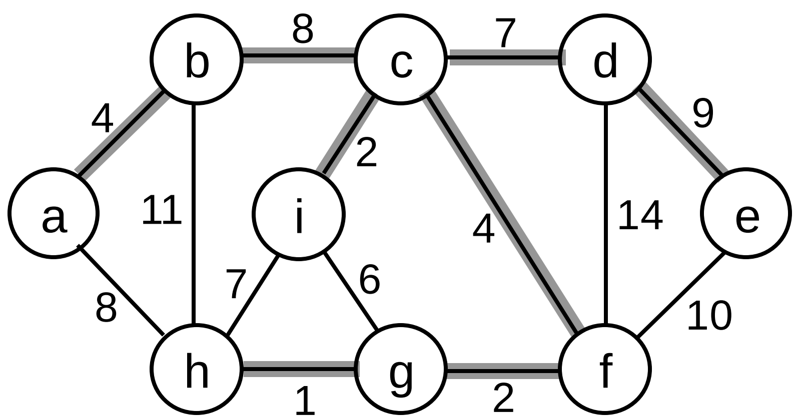
Li Jiang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- What is minimum spanning tree (MST)?
  - Definition and applications
- MST algorithms
  - A generic approach with theoretic proof
  - Prim's algorithm
  - Kruskal's algorithm

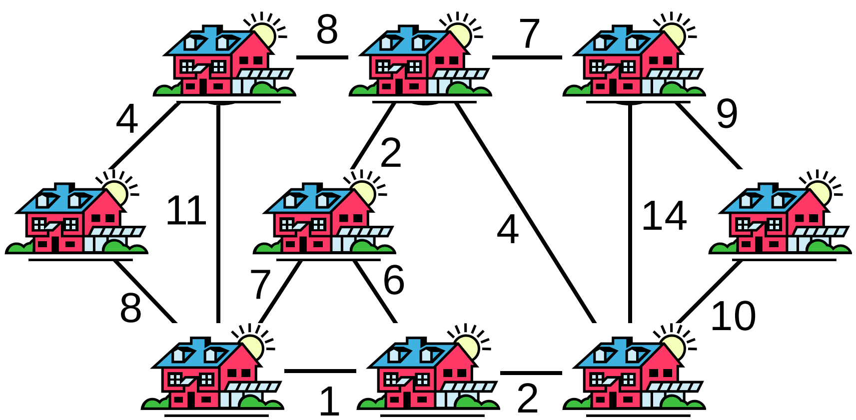
Minimum spanning tree

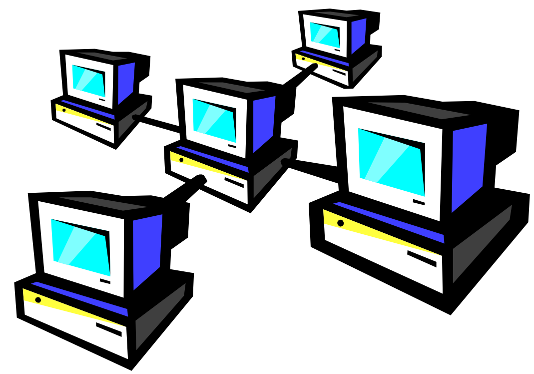
- Spanning tree
  - A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph
- Minimum spanning tree (MST)
  - Spanning tree with the minimum sum of weights
  - If a graph is not connected, then there is an MST for each connected component of the graph



Applications of MST

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





Problem setting

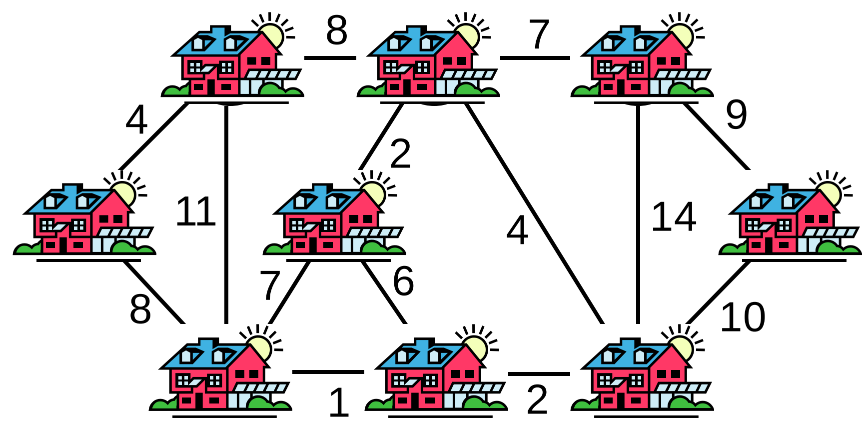
- A town has a set of houses and a set of roads
- A road connecting houses u and v has a cost w(u, v)

Minimum spanning trees (MSTs)

- Problem definition:
  - Given an undirected weighted graph, find a set of edges such that: (1) everyone stays connected and (2) the total cost is minimum

Find $T \subseteq E$ such that:

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

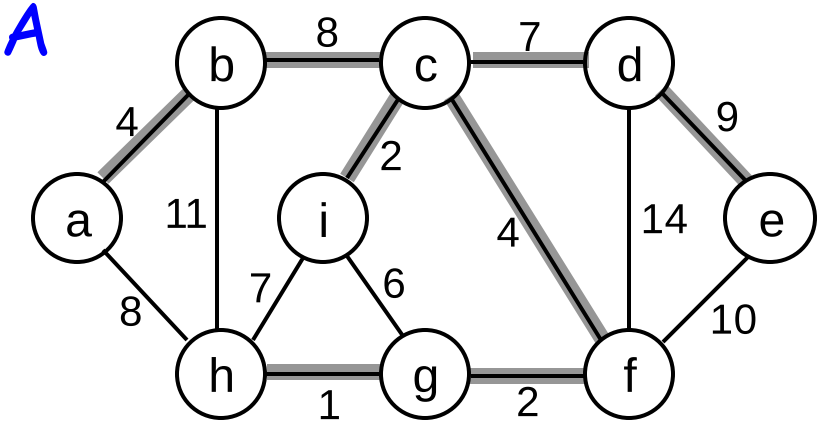


Properties of MST:

(1) MST is **not** unique; (2) MST has no cycles;

Growing an MST: generic approach

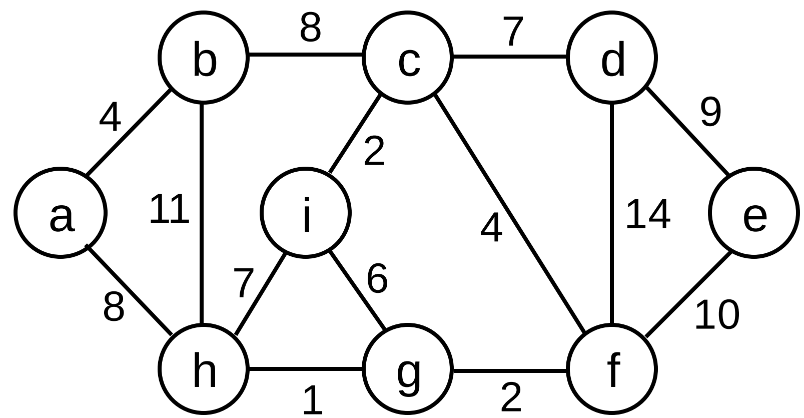
- Grow a set *A* of edges (initially empty)
- Incrementally add edges to *A* such that they would belong to an MST



- **Idea: add only "safe" edges**
  - An edge (u, v) is **safe** for *A*, if and only if $A \cup \{(u, v)\}$ is also a subset of **some** MST

Generic MST algorithm

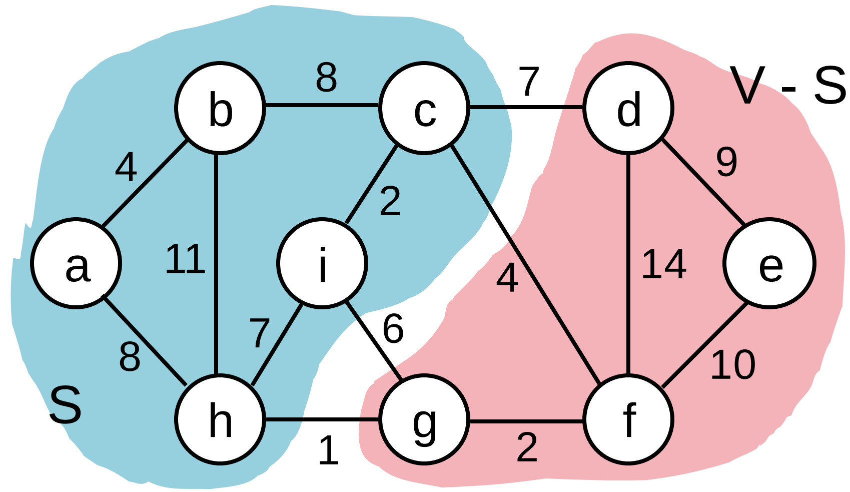
$A \leftarrow \varnothing$
while A is not a spanning tree
    do find an edge (u, v) that is safe for A
        $A \leftarrow A \cup \{(u, v)\}$
return A



- How do we find safe edges?

Finding safe edges

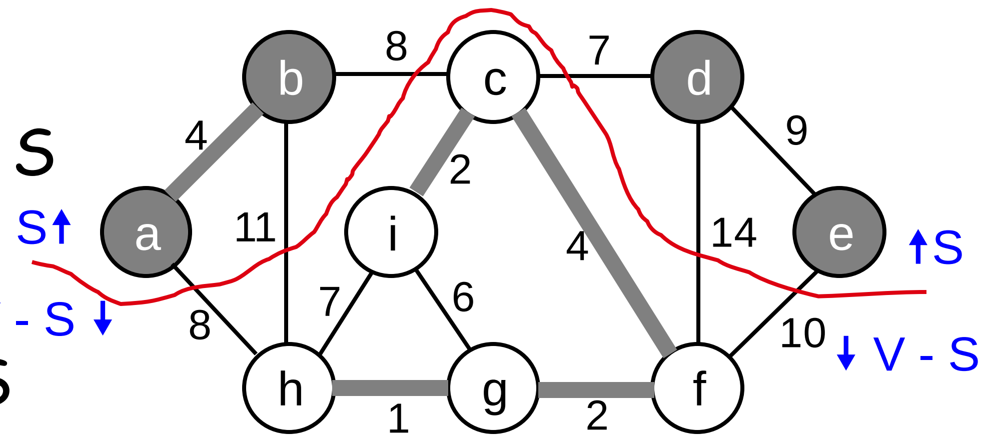
- Let's look at edge (h, g)
  - Is it safe for A initially?



- Yes. Why?
  - Let $S \subset V$ be any set of vertices that includes h but not g (so that g is in V - S)
  - In any MST, there has to be one edge (at least) that connects S with V - S
  - Why not choose the edge with minimum weight (h, g)?

Definitions

- A **cut** (S, V - S) is a partition of vertices into two disjoint sets S and V - S



- An edge **crosses** the cut (S, V - S) if one endpoint is in S and the other in V – S

- A cut **respects** a set A of edges ⇔ no edge in A crosses the cut

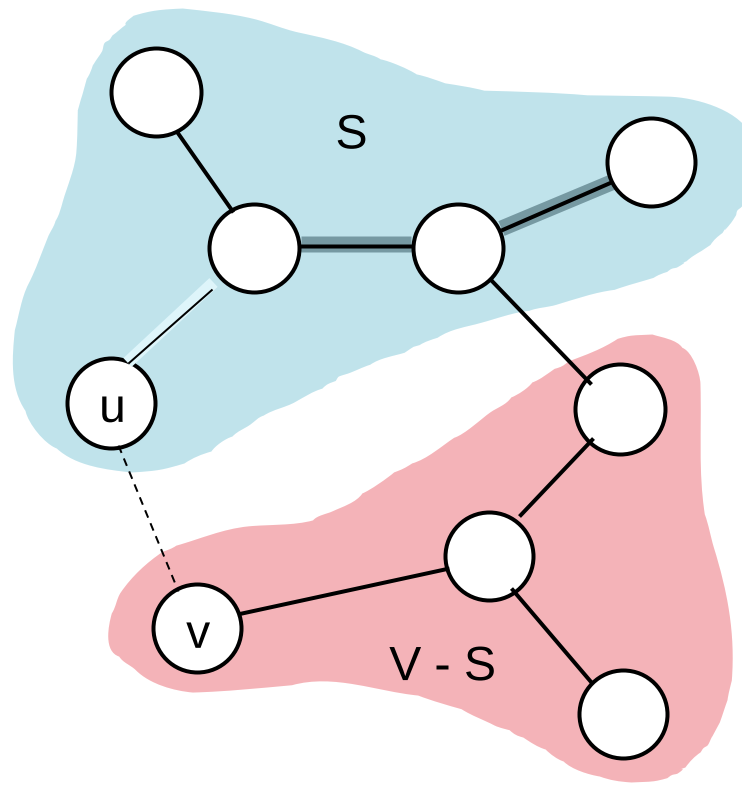
- An edge is a **light edge** crossing a cut ⇔ its weight is minimum over all edges crossing the cut
  - Note that for a given cut, there can be > 1 light edges crossing it

Theorem

- Let A be a subset of some MST, $(S, V - S)$ be a **cut** that respects A, and (u, v) be a **light edge** crossing $(S, V-S)$. Then (u, v) is safe for A.

Proof:

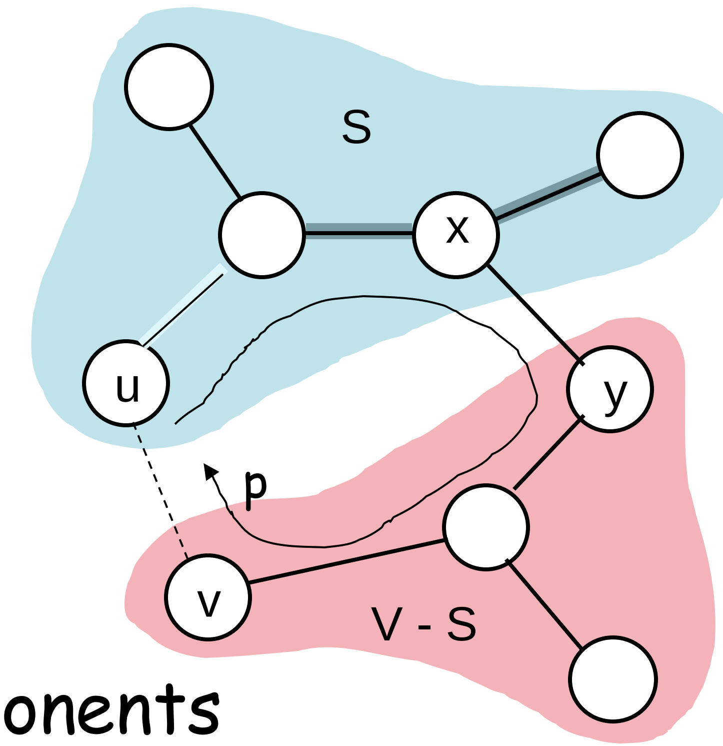
- Let T be an MST that includes A
  - edges in A are shaded
- <u>Case1:</u> If T includes (u,v), then it would be safe for A
- <u>Case2:</u> Suppose T does not include the edge (u, v)
  - **Idea**: construct another MST T' that includes $A + \{(u, v)\}$



Theorem: proof

- T contains a unique path p between u and v
- Path p must cross the cut (S, V - S) at least once: let (x, y) be that edge
- Let's remove (x, y) $\Rightarrow$ breaks T into two components
- Adding (u, v) reconnects the components

  $T' = T - \{(x, y)\} + \{(u, v)\}$

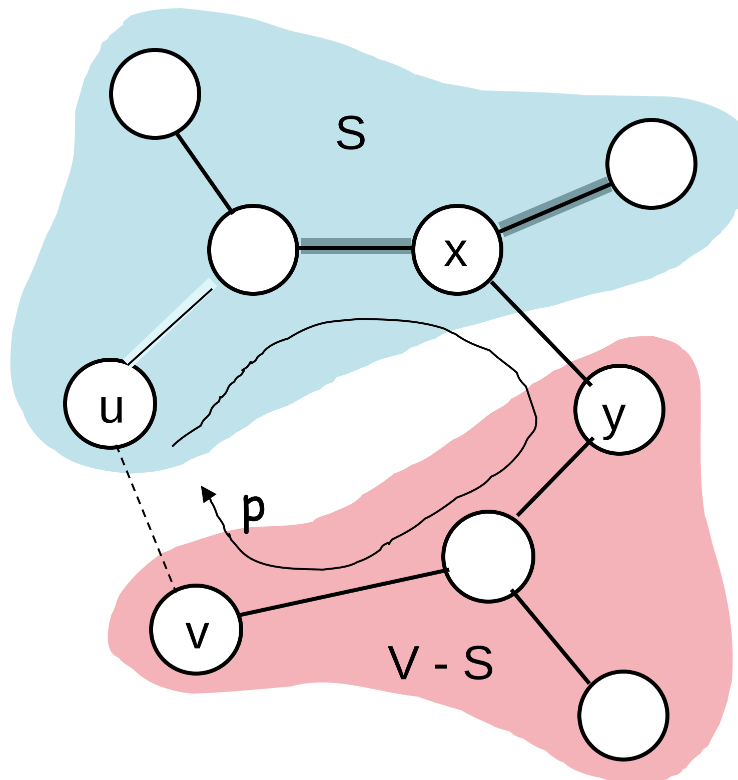


Theorem: proof

$T' = T - \{(x, y)\} + \{(u, v)\}$

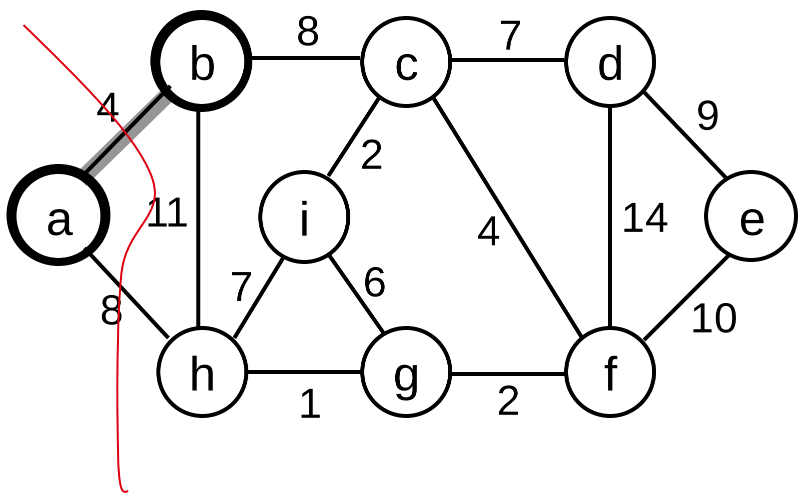
Have to show that T' is an MST:

- (u, v) is a light edge
  $\Rightarrow w(u, v) \leq w(x, y)$
- $w(T') = w(T) - w(x, y) + w(u, v)$
  $\leq w(T)$
- Since T is an MST,
  $w(T) \leq w(T') \Rightarrow T'$ must be an MST as well



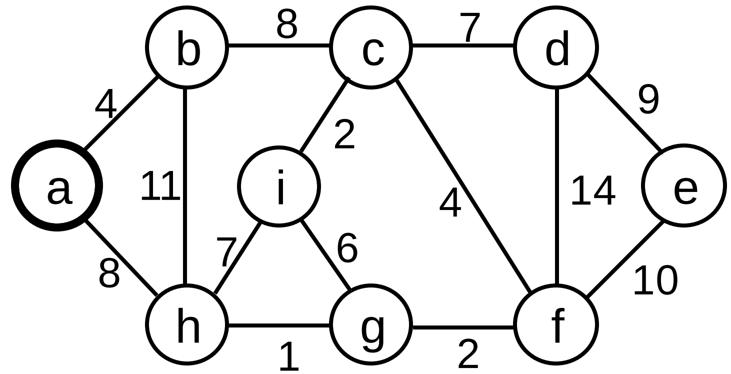
Prim's algorithm

- The edges in set A always <u>form a single tree</u>
- Starts from an arbitrary "root": $V_A = \{a\}$
- At each step:
  - Find a light edge crossing $(V_A, V - V_A)$
  - Add this edge to A
  - Repeat until the tree spans all vertices



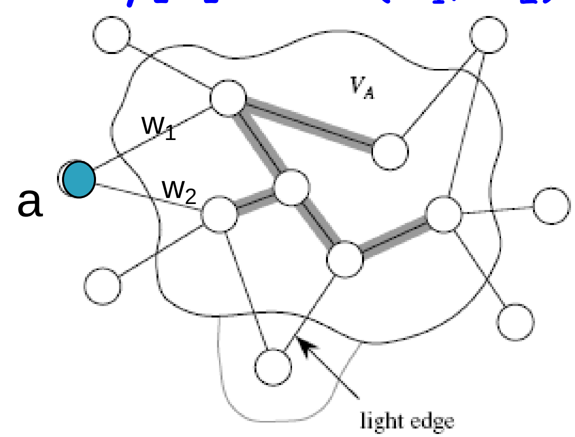
How to find light edges quickly?

- Use a priority queue Q to include vertices not in the tree, i.e., $(V - V_A)$
  - $V_A = \{a\}$, $Q = \{b, c, d, e, f, g, h, i\}$



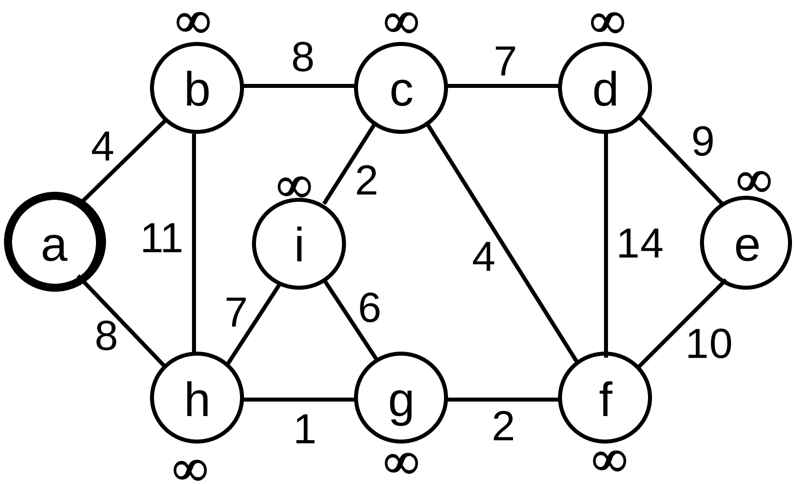
- Associate a key to each vertex v in Q:
  - key[v] = minimum weight of any edge (u, v) connecting v to V_A
  - key[v] = ∞, if v is not adjacent to any vertices in V_A

key[a] = min(w_1, w_2)



- After adding a new vertex to V_A, update the weights of all vertices <u>adjacent to it</u>
  - E.g., after adding a, k[b]=4 and k[h]=8

Example

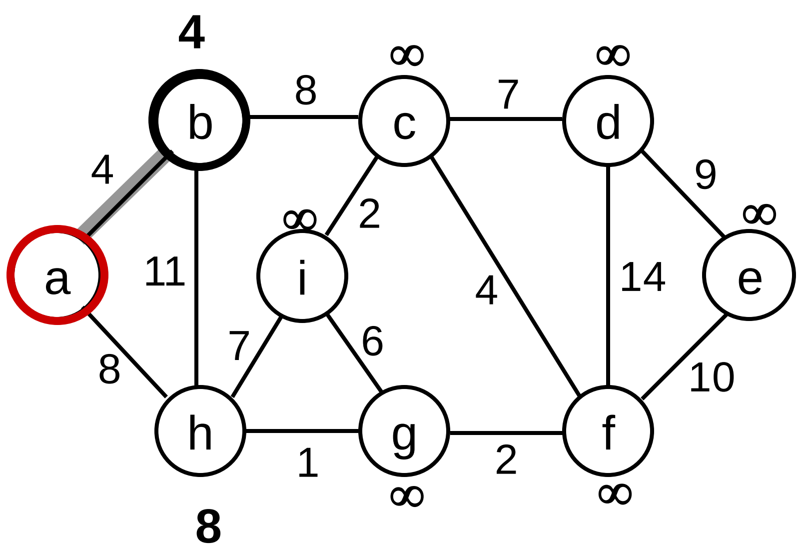


$0\ \ \infty\ \infty\ \ \infty\ \infty\ \infty\ \infty\ \infty\ \infty$

$Q = \{a, b, c, d, e, f, g, h, i\}$

$V_A = \varnothing$

Extract-MIN(Q) $\Rightarrow$ a



key [b] = 4 π [b] = a

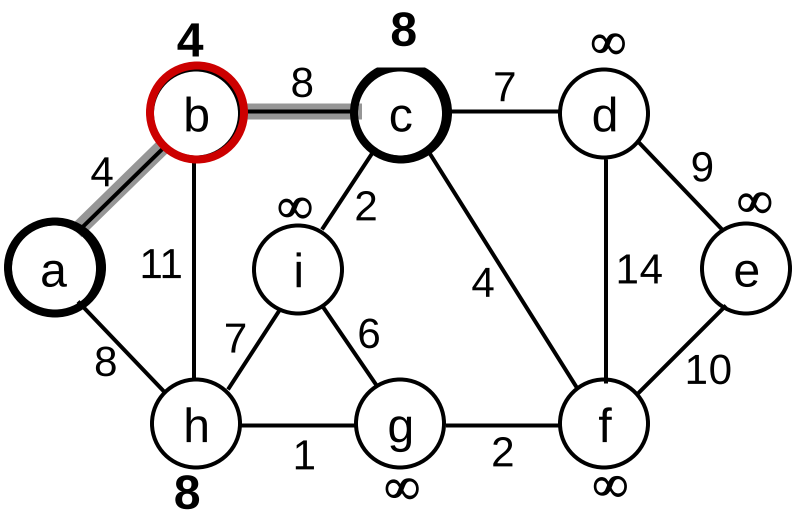
key [h] = 8 π [h] = a

4 $\infty\ \infty\ \infty\ \infty\ \infty$ **8** ∞

$Q = \{b, c, d, e, f, g, h, i\}$ $V_A = \{a\}$

Extract-MIN(Q) $\Rightarrow$ b

Example



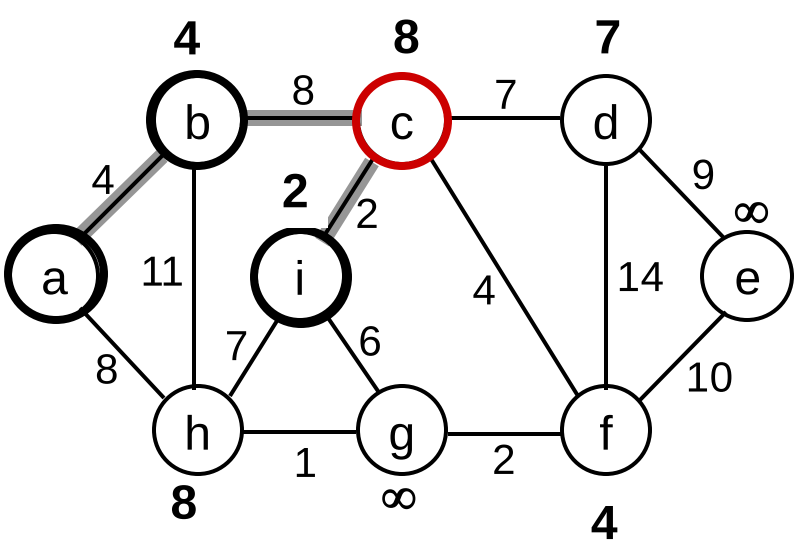
key [c] = 8 π [c] = b

key [h] = 8 π [h] = a unchanged

8 ∞ ∞ ∞ ∞ **8** ∞

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

Extract-MIN(Q) ⇒ c



key [d] = 7 π [d] = c

key [f] = 4 π [f] = c

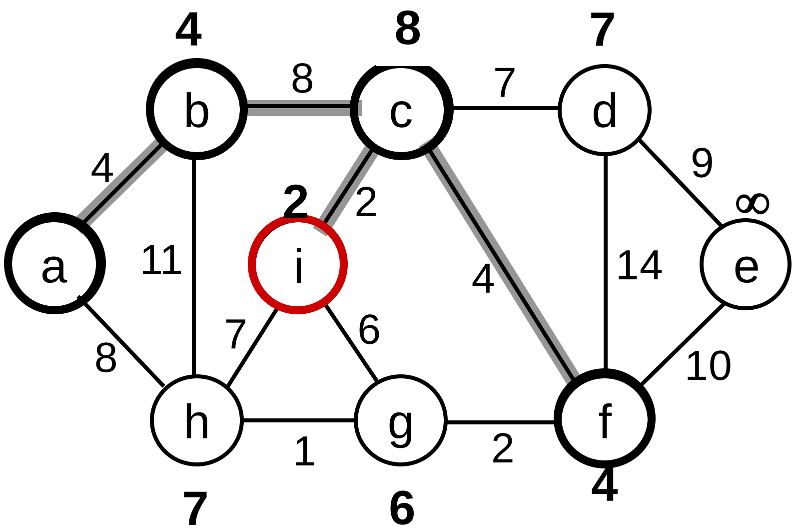
key [i] = 2 π [i] = c

7 ∞ **4** ∞ **8** **2**

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

Extract-MIN(Q) ⇒ i

Example



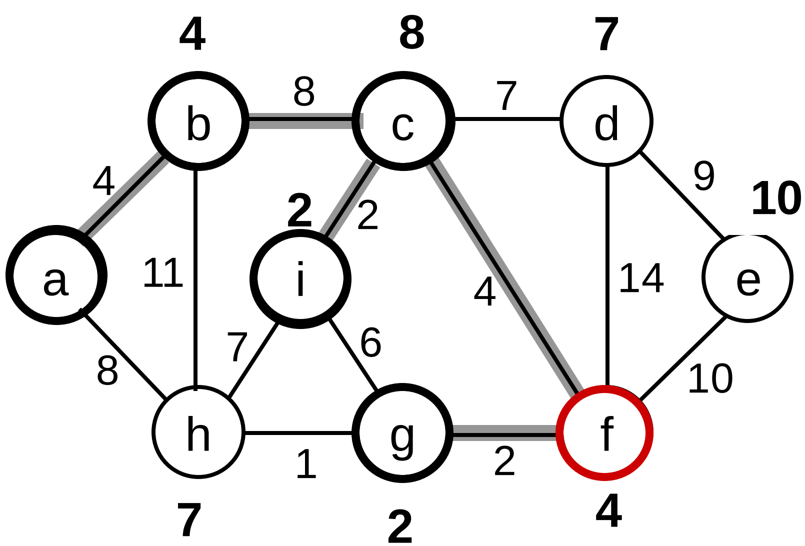
key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 7

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

Extract-MIN(Q) $\Rightarrow$ f



key [g] = 2 π [g] = f

key [d] = 7 π [d] = c unchanged

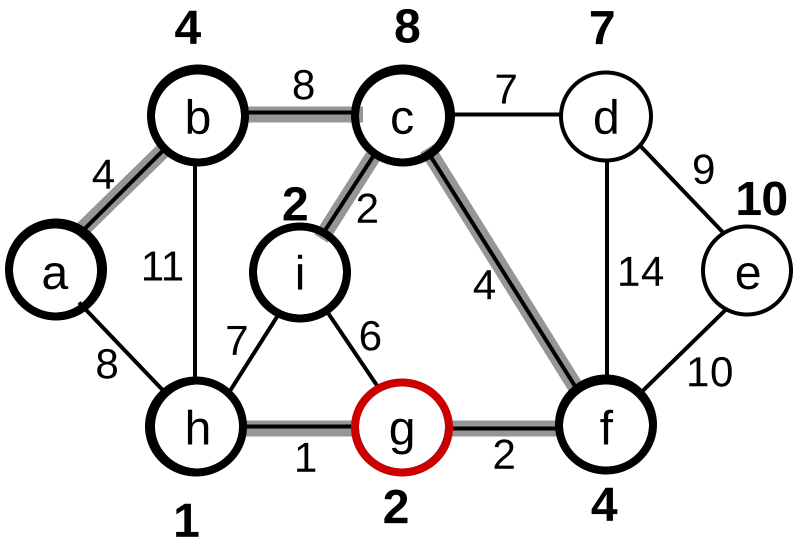
key [e] = 10 π [e] = f

7 10 2 7

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

Extract-MIN(Q) $\Rightarrow$ g

Example

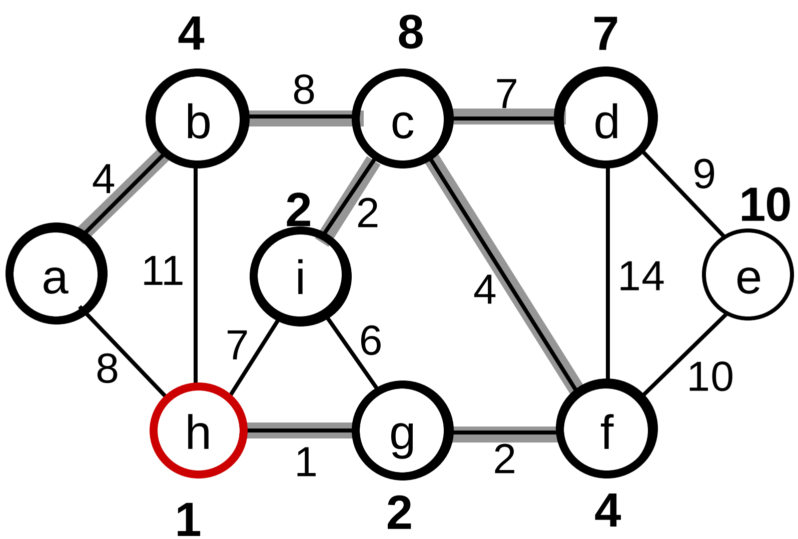


key [h] = 1 π [h] = g

7 10 1

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

Extract-MIN(Q) ⇒ h

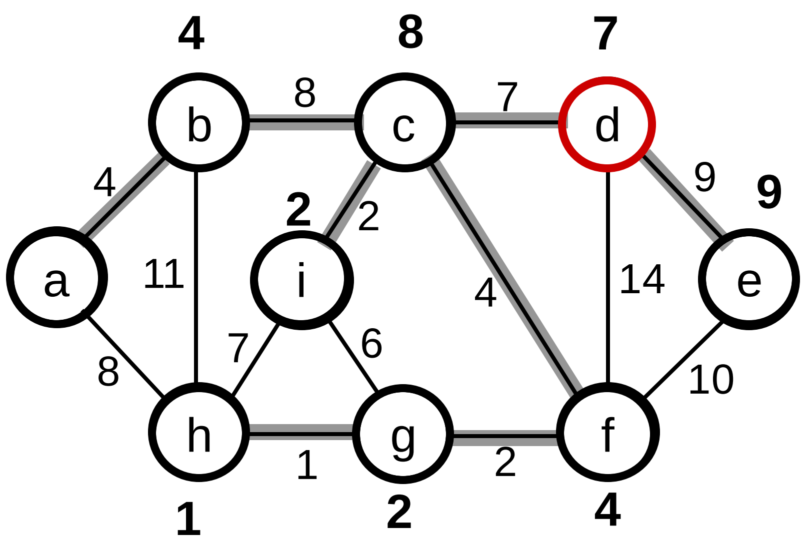


7 10

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

Extract-MIN(Q) ⇒ d

Example



key [e] = 9 π [e] = d

9

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

Extract-MIN(Q) $\Rightarrow$ e

Q = $\varnothing$ V_A = {a, b, c, i, f, g, h, d, e}

Prim(V, E, w, r)

Total time: O(VlogV + ElogV) = O(ElogV)

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

Lines 1–5: O(V) if Q is implemented as a min-heap

Line 6: O(logV)

Line 7: Executed |V| times

Line 8: Takes O(logV)

Lines 7–8: Min-heap operations: O(VlogV)

Line 9: Executed O(E) times

Line 10: Constant

Lines 11–12: Takes O(logV)

Lines 9–12: O(ElogV)

Prim(V, E, w, r)

```
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
       π[u] ← NIL
       INSERT(Q, u)
DECREASE-KEY(Q, r, 0)          ▶ key[r] ← 0
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       for (j=0; j<|V|; j++)
          if (A[u][j]=1)
             if v ∈ Q and w(u, v) < key[v]
                then π[v] ← u
                     DECREASE-KEY(Q, v, w(u, v))
```

Total time: $O(V\log V + E\log V + V^2) = O(E\log V + V^2)$

$O(V)$ if Q is implemented as a min-heap

key[r] ← 0 ← $O(\log V)$

Executed |V| times

Takes $O(\log V)$

Min-heap operations: $O(V\log V)$

Executed $O(V^2)$ times total

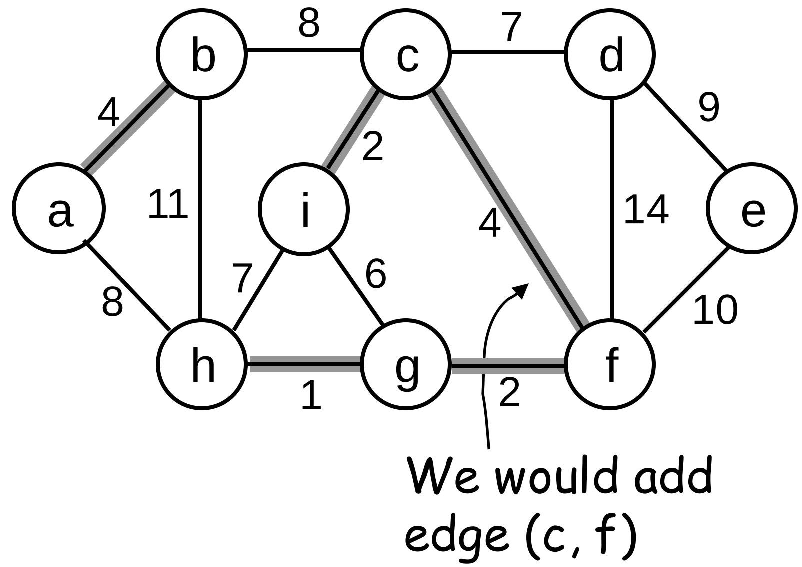
Constant

Takes $O(\log V)$

$O(E\log V)$

Kruskal's algorithm

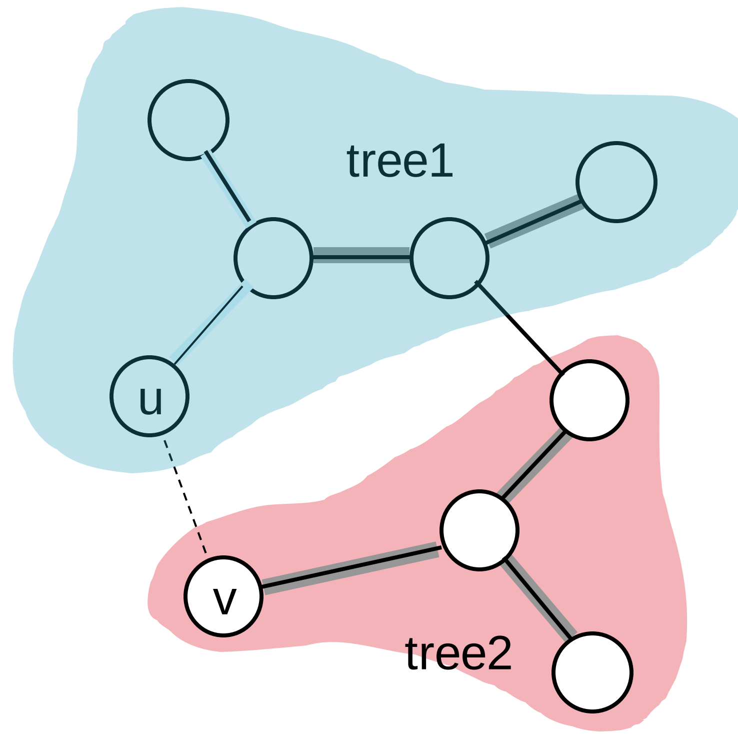
- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light** edge that connects them



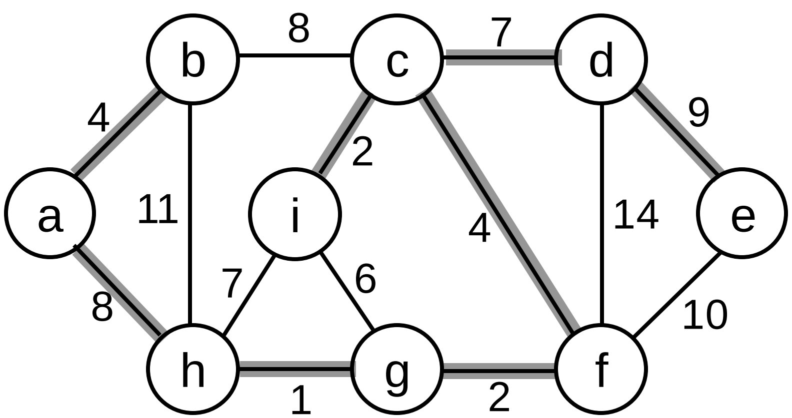
- Which components to consider at each iteration?
  - Scan the set of edges in monotonically increasing order by weight

Kruskal's algorithm

- How is it different from Prim's algorithm?
  - Prim's algorithm grows one tree all the time
  - Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time
  - Trees are merged together using **safe** edges



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

Algorithm 1: a naive method

Assume vertices are 1, 2, …, n, and E >= V

```
Sort all the edges                          ← O(ElogE)
for each v ∈ V                              } O(V)
    setArray[v] = {v}
for each edge (u, v) ∈ E
    setU = setArray[u], setV = setArray[v]  } O(1)
    isConnected = false
    for each vertex w ∈ setU                } O(setU.size)
        if w == v
            isConnected = true
            break
    if isConnected == false                 } O(setV.size)
        newSet = setU ∪ setV
        setArray[u] = setArray[v] = newSet
        R = R ∪ {(u, v)}
Output R
```

O(VE)

Can we do better?

Algorithm 2: using labels

- Using labels
  - A label means a connected component
  - Assign a unique label to each vertex initially
  - When merging two connected components, we always change the labels of vertices in the small component to the label of the large component
    - The cost of changing labels is smaller

Algorithm 2: using labels

Assume vertices are 1, 2, …, n, and E >= V

```
Sort all the edges  ← O(ElogE)
for each v ∈ V
    label[v] = v                          } O(V)
    setArray[v] = {v}
for each edge (u, v) ∈ E
    uL = label[u], vL = label[v]
    if uL == vL  continue                 } O(E)
    R.add((u, v))
    if setArray[uL].size >= setArray[vL].size
        for each vertex w ∈ setArray[vL]
            label[w] = uL                 } O(setArray[vL].size)
            setArray[uL].add(w)
    else
        for each vertex w ∈ setArray[uL]
            label[w] = vL                 } O(setArray[uL].size)
            setArray[vL].add(w)
Output R
                                          } ???
```

What's the total times of changing the labels for all the vertices?

A vertex's label is changed at most O(logV) times:

- Fix a vertex **x**
- If x is in a set **S** and its label changes, then **S** is merged to another set whose updated size is at least 2|**S**|
  - Initially, x's set has size 1 (containing itself)
  - After merging for x's set once, it has size at least 2
  - After merging for x's set twice, it has size at least 4
  - After merging for x's set three times, it has size at least 8
  - …
  - After merging for x's set k times, it has size at least 2^k
- Since the total number of vertices is V, the number of times for changing labels for x is at most k = O(logV)

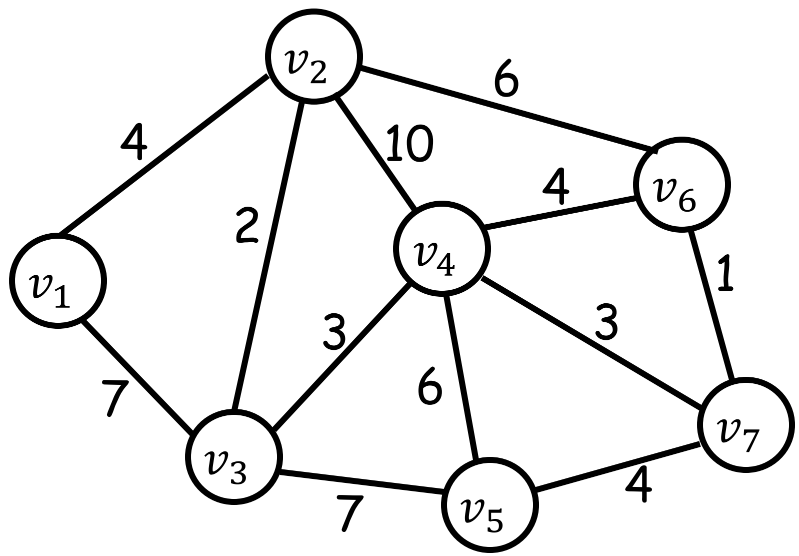
Algorithm 2: using labels

Assume vertices are 1, 2, …, n, and E >= V

```
Sort all the edges  ←  O(ElogE)
for each v ∈ V
    label[v] = v                                        O(V)
    setArray[v] = {v}
for each edge (u, v) ∈ E
    uL = label[u], vL = label[v]
    if uL == vL  continue                               O(E)
    R.add((u, v))
    if setArray[uL].size >= setArray[vL].size                        O(ElogE)
        for each vertex w ∈ setArray[vL]
            label[w] = uL
            setArray[uL].add(w)
    else                                                O(VlogV)
        for each vertex w ∈ setArray[uL]
            label[w] = vL
            setArray[vL].add(w)
Output R
```

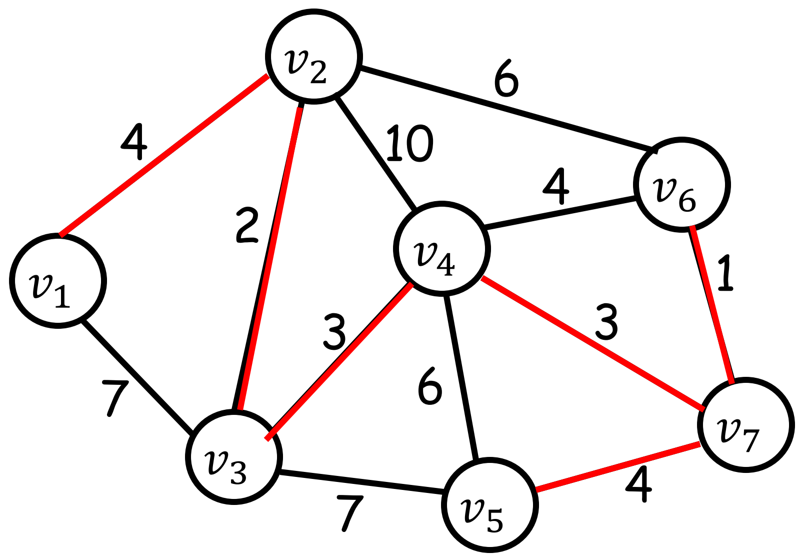
Exercises

- Given the following graph, find its MST using Prim's algorithm and Kruskal's algorithm respectively



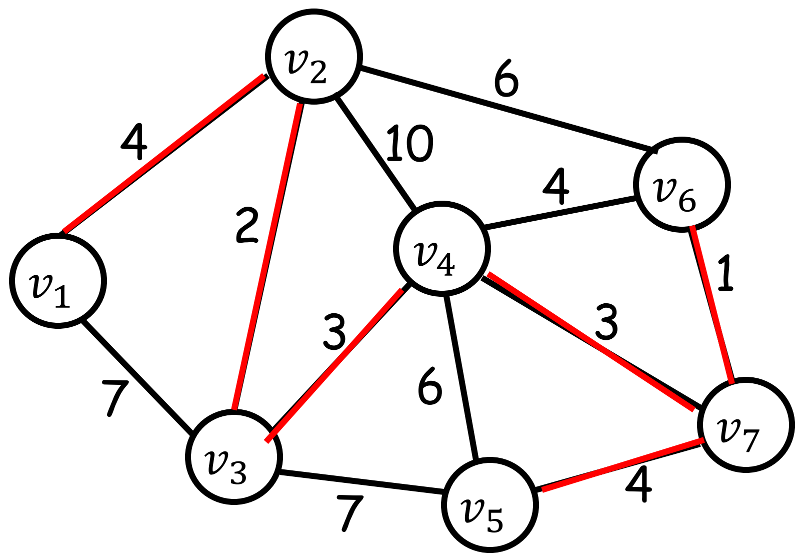
Exercises

- Given the following graph, find its MST using Prim's algorithm and Kruskal's algorithm respectively



Exercises

- Given the following graph, find its MST using Prim's algorithm and Kruskal's algorithm respectively



Recommended reading

- Reading materials
  - Textbook Chapter 23

- Next lecture
  - Shortest paths, Chapters 24&25