Algorithms and Their Applications
- Greedy Algorithms -

Won-Yong Shin

May 25th, 2020

연세대학교
YONSEI UNIVERSITY

- Greedy technique
  - Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:
    - *Feasible*
    - *Locally optimal*
    - *Irrevocable* (i.e., once made, it cannot be changed on subsequent steps)
  - For some problems, yields an **optimal solution** for every instance
  - For most, *does not* but can be useful for fast approximations

- ***Optimal*** solutions:
  - Change making for "*normal*" coin denominations
  - Minimum spanning tree (MST)
  - Single-source shortest paths
  - Simple scheduling problems

- ***Approximations***:
  - Traveling salesman problem (TSP)
  - Knapsack problem
  - Other combinatorial optimization problems

Toy Example: Change-Making Problem

- Change-making problem
  - Given unlimited amounts of coins of denominations $d_1 > d_2 \ldots > d_m$, give change for amount n with the *least number* of coins

- Example
  - $d_1 = 25$ (quarter), $d_2 = 10$ (dime), $d_3 = 5$ (nickel), $d_4 = 1$ (penny) and $n = 48$
  - Greedy solution:
    - 1 quarter, 2 dimes, and 3 pennies

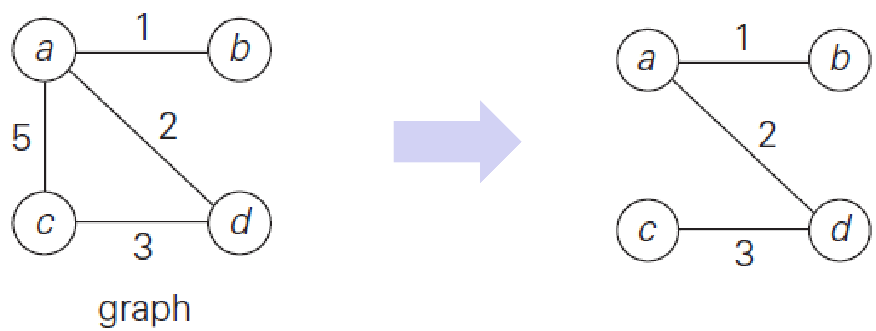
- Greedy solution is
  - optimal for "*normal*" set of denominations
  - <u>may not</u> be optimal for arbitrary coin denominations
    - E.g., $d_1 = 25$, $d_2 = 10$, $d_3 = 1$ and $n = 30$

Minimum Spanning Tree (MST)

- *Spanning tree* of an undirected connected graph G
  - A connected *acyclic* subgraph of G that includes **all of G's vertices**

- *Minimum spanning tree* (MST) of a weighted, connected graph G
  - A spanning tree of G of **minimum** total weight
  - E.g., telecommunications in cable-line infrastructure

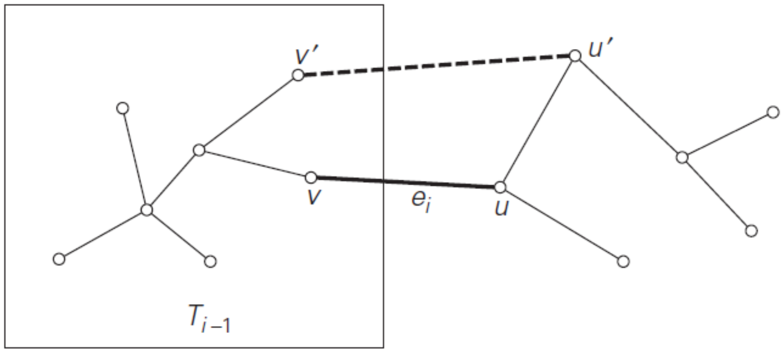
- Example:

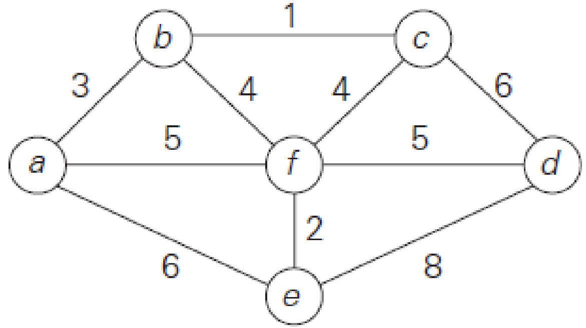


- Two algorithms for finding an MST for a weighted graph
  - Prim's algorithm
  - Kruskal's algorithm

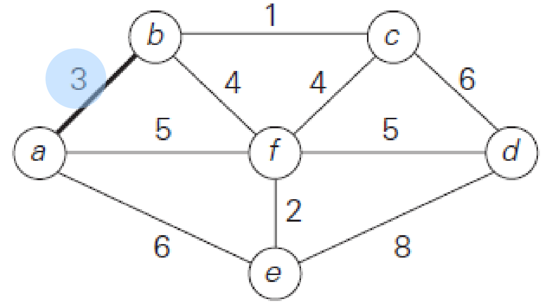
- Operations:
  - Start with tree T_1 consisting of *one* (any) vertex and "grow" tree *one vertex at a time* to produce MST through a series of expanding subtrees $T_1, T_2, \ldots, T_n$
  - On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a "greedy" step!)
  - Stop when **all vertices** are included

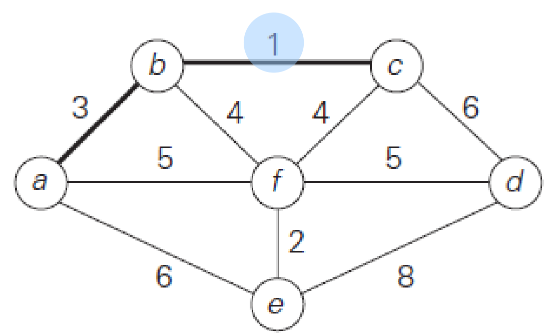
- Correctness proof (by *induction*)
  - Does Prim's algorithm always yield an MST? → **yes**
  - Basis of the induction: T_0 consists of a single vertex
  - Inductive step:
    - Assume that $T_{i\text{-}1}$ is a part of some MST
    - Start from **contradiction**
    - A cycle must be formed by adding e_i





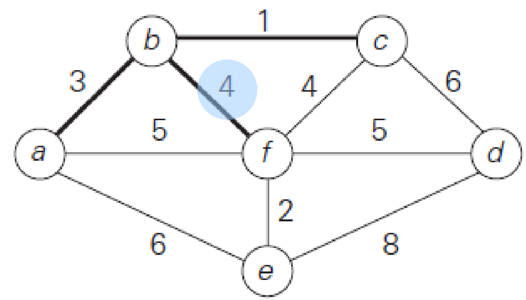
Tree vertices	**Remaining vertices**
a(–, –)	**b(a, 3)** c(–, ∞) d(–, ∞) e(a, 6) f(a, 5)
b(a, 3)	**c(b, 1)** d(–, ∞) e(a, 6) f(b, 4)



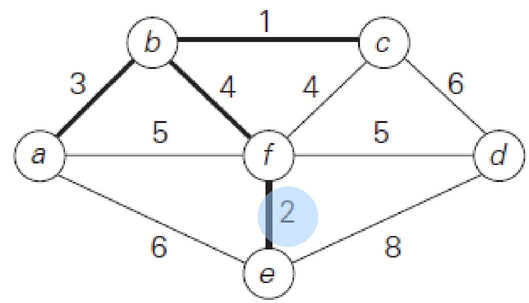


Example of Prim's Algorithm (2/3)

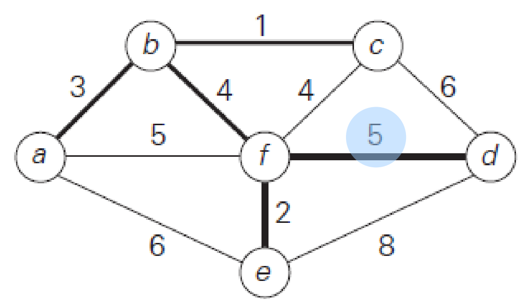
Tree vertices	Remaining vertices
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**



Tree vertices	Remaining vertices
f(b, 4)	d(f, 5) **e(f, 2)**



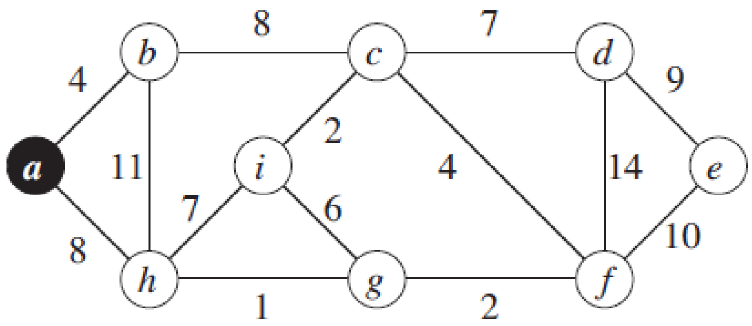
Tree vertices	Remaining vertices
e(f, 2)	**d(f, 5)**

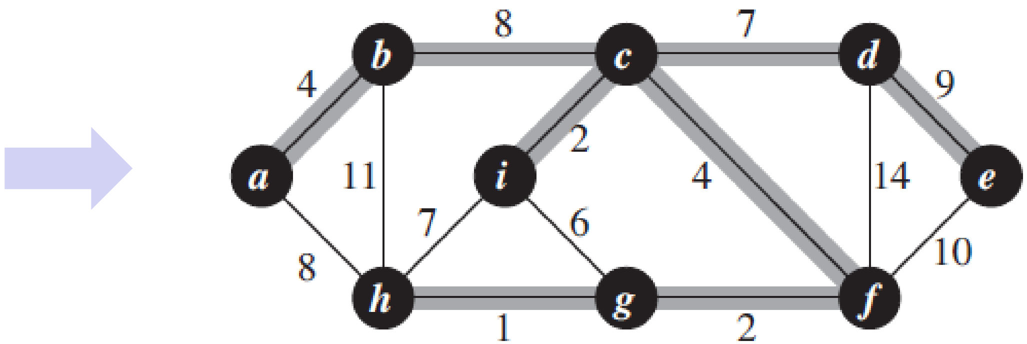


Tree vertices	Remaining vertices
d(f, 5)	

cf. priority queue

Example of Prim's Algorithm (3/3)





Prim's Algorithm – Time Efficiency (1/2)

- Assumption 1
  - The case where a graph is represented by its weight matrix the priority queue is implemented as an *unordered* array

➡ **Efficiency:** $\Theta(|V|^2)$

- Assumption 2
  - The case where a graph is represented by its adjacency lists the priority queue is implemented as a ***min-heap***

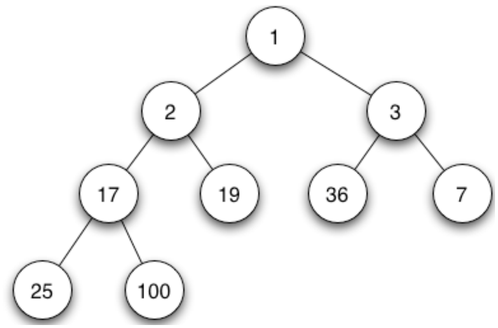
MST-PRIM(G, w, r)

```
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

O(*V*)

O(*V* log*V*)

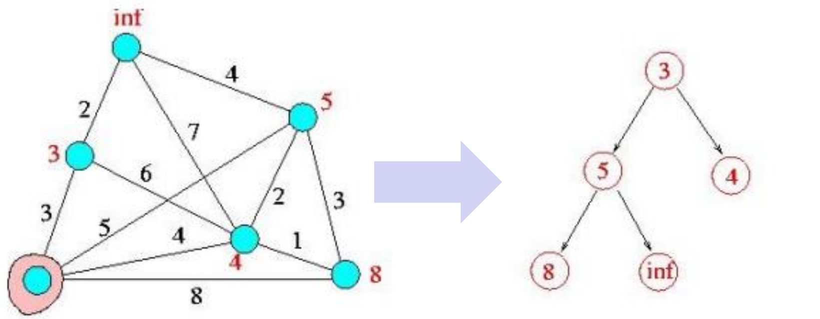
O(*E* log*V*)



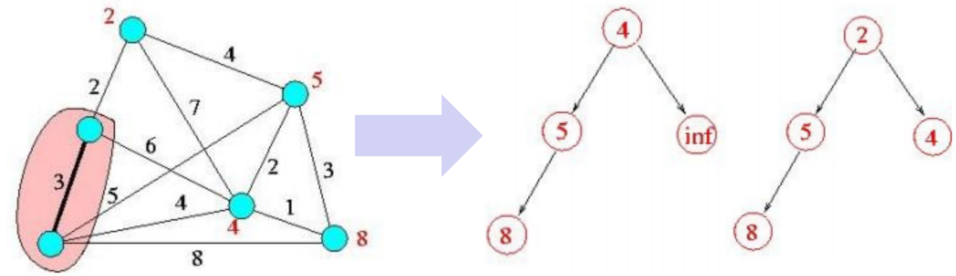
Binary min-heap

Efficiency: $(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$

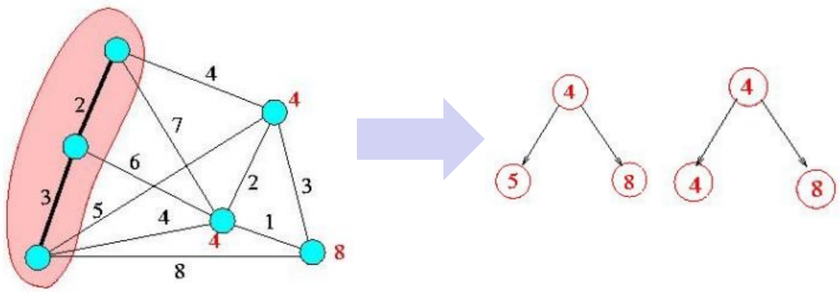
Using min-heap in Prim's Algorithm



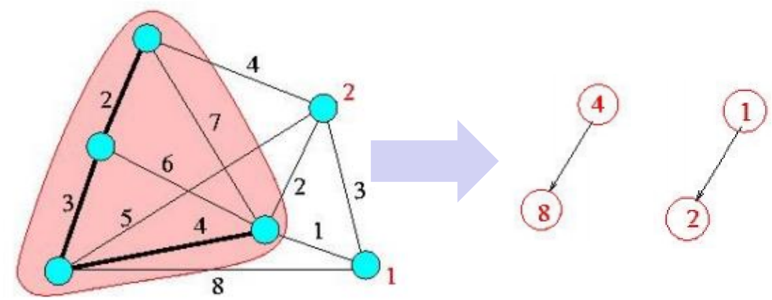
Step 1



Step 2



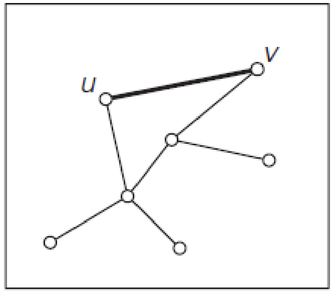
Step 3



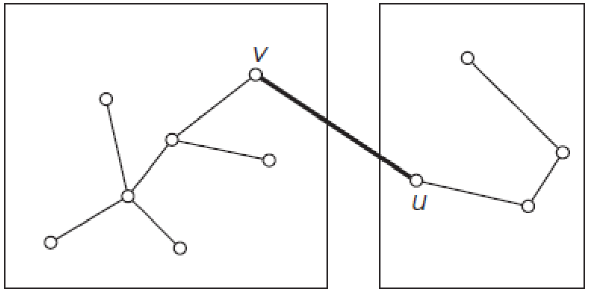
Step 4

- Operations:
  - Sort the *edges in nondecreasing order* of lengths
  - "Grow" tree *one edge at a time* to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
  - On each iteration, add the next edge on the sorted list unless this would create a **cycle**
  - (If it would, skip the edge)

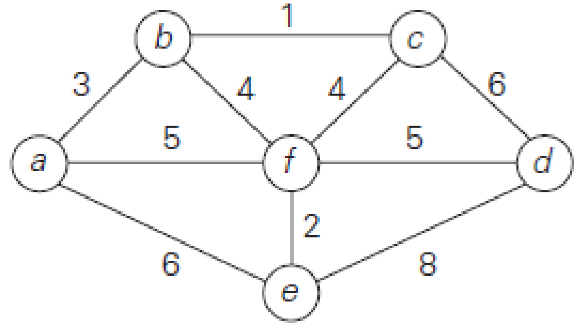
- Observation
  - A new cycle is created iff the new edge connects two vertices already connected by a path (i.e., the two vertices belong to the same connected component)



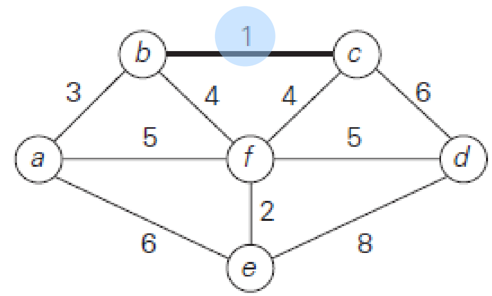
Creating a cycle

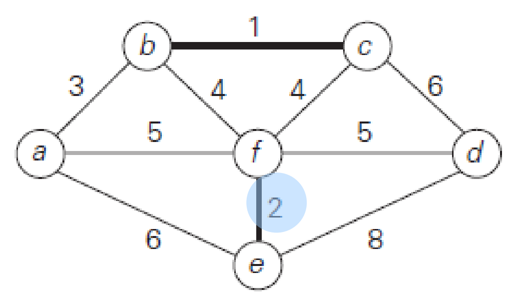


Not creating a cycle

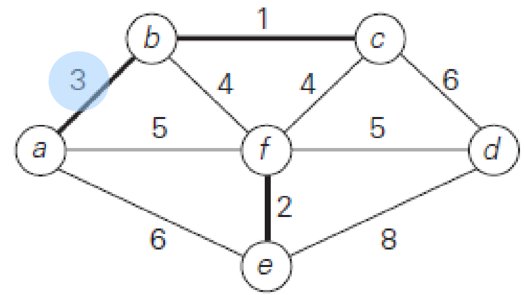


Tree edges	Sorted list of edges
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8

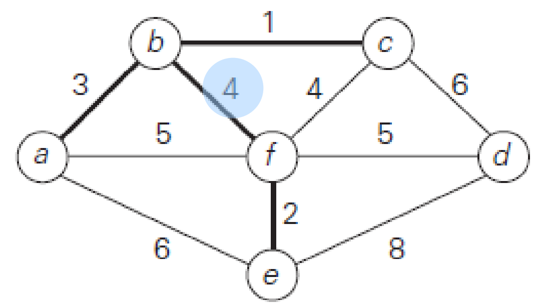




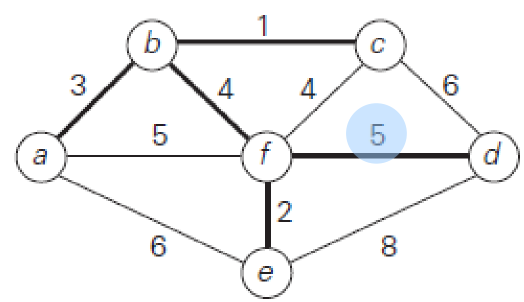
Tree edges	Sorted list of edges									
ef	bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2	1	2	3	4	4	5	5	6	6	8



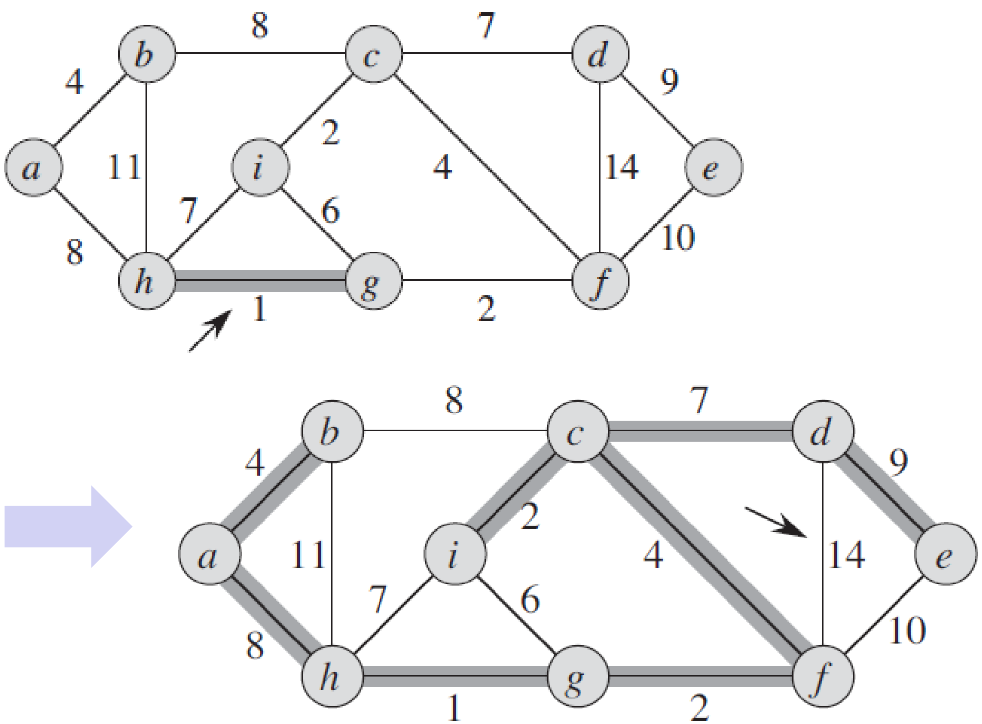
Tree edges	Sorted list of edges									
ab	bc	ef	ab	**bf**	cf	af	df	ae	cd	de
3	1	2	3	4	4	5	5	6	6	8



Tree edges	Sorted list of edges									
bf	bc	ef	ab	bf	cf	af	**df**	ae	cd	de
4	1	2	3	4	4	5	5	6	6	8



df
5

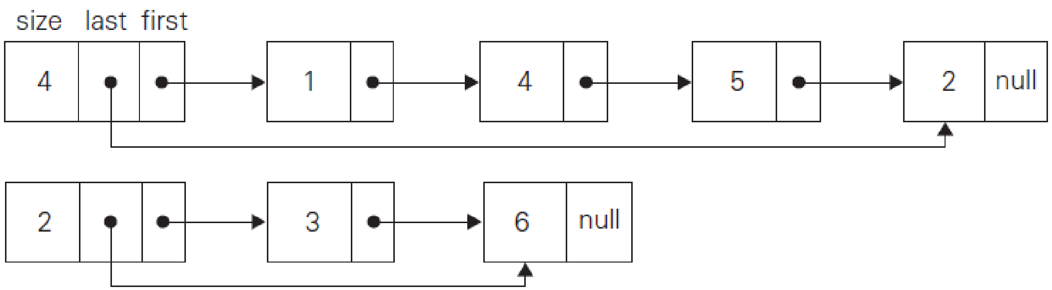


a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 8 10 1 2
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 8 10 1 2

Notes About Kruskal's Algorithm

- Some notes
  - Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)
  - Cycle checking
  - *Union-find* algorithms
    - Check whether two vertices belong to the same tree

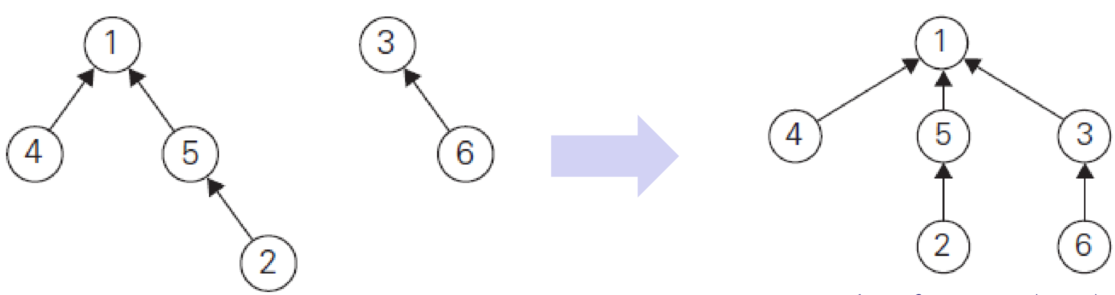
quick find



subset representatives

element index	representative
1	1
2	1
3	3
4	1
5	1
6	3

quick union



Result of *union*(5, 6)

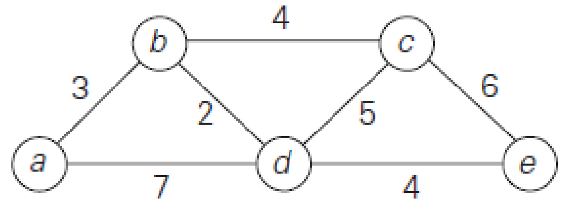
- *Single source shortest paths problem*
  - Given a weighted connected graph G, find shortest paths from source vertex s to each of the other vertices

- *Dijkstra's algorithm*
  - Similar to Prim's MST algorithm, with a different way of computing numerical labels
  - Among vertices not already in the tree, it finds vertex u with the smallest sum

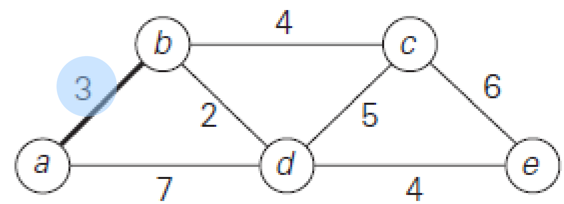
$$d_v + w(v,u)$$

    - v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree)
    - d_v is the length of the shortest path form **source** to v
    - $w(v,u)$ is the length (weight) of edge from v to u

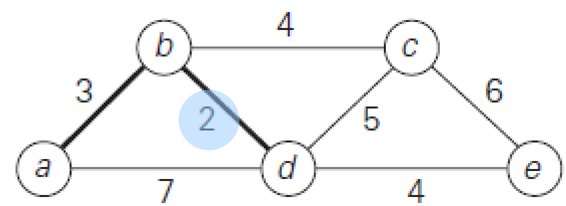
Example of Dijkstra's Algorithm (1/2)



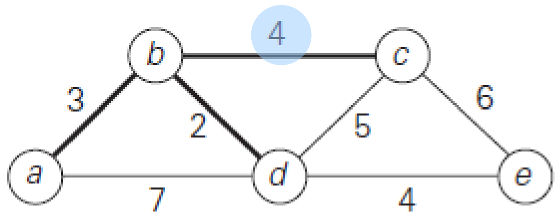
Tree vertices	**Remaining vertices**
a(−, 0)	**b**(**a**, **3**) c(−, ∞) d(a, 7) e(−, ∞)

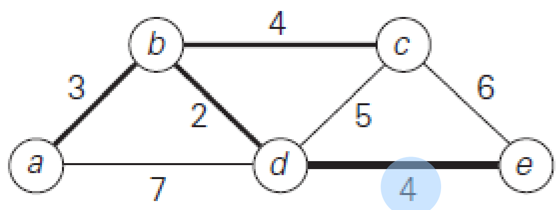


b(a, 3)	c(b, 3 + 4) **d**(**b**, **3 + 2**) e(−, ∞)



Tree vertices	Remaining vertices
d(b, 5)	**c**(**b**, **7**) e(d, 5 + 4)
c(b, 7)	**e**(**d**, **9**)
e(d, 9)	





- Assumption 1
  - The case where a graph is represented by its weight matrix and the priority queue is implemented as an *unordered* array

    Efficiency: $\Theta(|V|^2)$

- Assumption 2
  - The case where a graph is represented by its adjacency lists the priority queue is implemented as a ***min-heap***

    Efficiency: $(|V| - 1 + |E|)O(\log|V|) = O(|E| \log |V|)$