



# **Bölüm 12: Açgözlü Algoritmalar**

## **Algoritmalar**



# Adım Adım Kazananlar

- Açgözlü algoritmalar, her adımda en iyi görünen seçeneği seçerek ilerler.
- Tıpkı bir define arayan gibi,
  - her aşamada en fazla kazancı sağlayacak seçeneği tercih eder.





# Açgözlü Algoritmalar

- Açgözlü algoritmaları, basamaklı bir yaklaşıma benzer.
- Her adımda, mevcut durum için en iyi sonucu verecek seçenek seçilir.
- Tıpkı bir yolculuk planlarken en kısa yolu seçmek gibi..
- Problemi bütün olarak ele almaz, mevcut en iyi seçeneğe odaklanır.
- Karmaşık problemleri çözmek için hızlı ve basit bir yöntem sunar.
- Kod yapısı genellikle basit ve anlaşılması kolaydır.
- En iyi (optimal) çözümü bulmayı garanti etmez.
- Çoğu durumda iyi bir yaklaşım üretir.



# Boruvka'nın Algoritması

- Çizge teorisinde kullanılan bir en küçük ağaç bulma algoritmasıdır.
- Başlangıçta her bir düğüm bir alt ağaç oluşturur.
- Her adımda,
  - her alt ağacın en az maliyetli kenarı bulunur ve
  - bu kenarlar birleştirilerek yeni bir ağaç oluşturulur.
- Greedy (açgözlü): Her adımda yerel olarak en iyi seçeneği seçer.
- Her adımda en az bir kenar seçilir ve ağaç büyütülür.
- Çalışma zamanı:  $O(E \log V)$ ,  $E$  kenar sayısı,  $V$  düğüm sayısı.

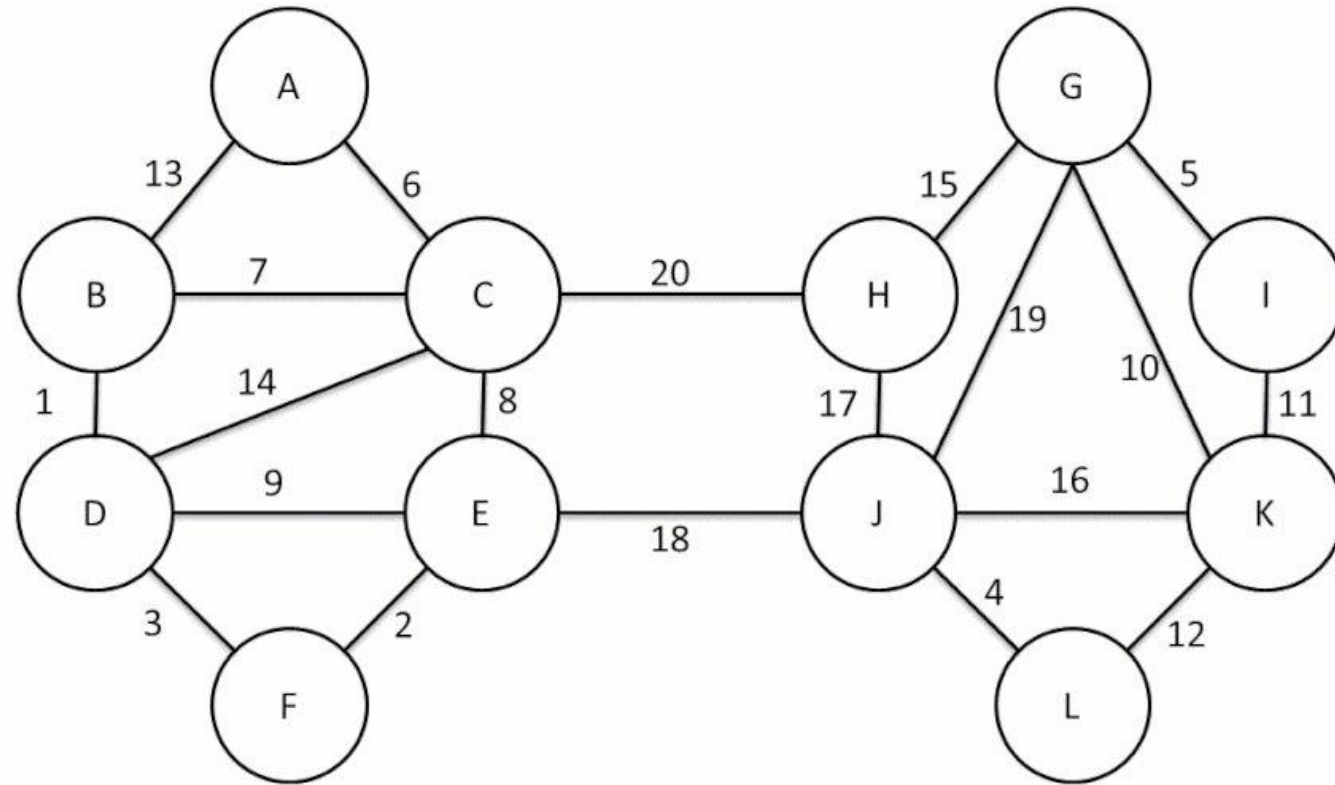


# Boruvka'nın Algoritması

- Adım 1: Her düğüm bir alt ağaç olarak başlar.
- Adım 2: Her alt ağaçtan, en az maliyetli kenarı içeren kenar seçilir.
- Adım 3: Seçilen kenarlarla alt ağaçlar birleştirilir.
- Adım 4: Aynı süreç, alt ağaçlar tek bir ağaçta toplanana kadar tekrarlanır.

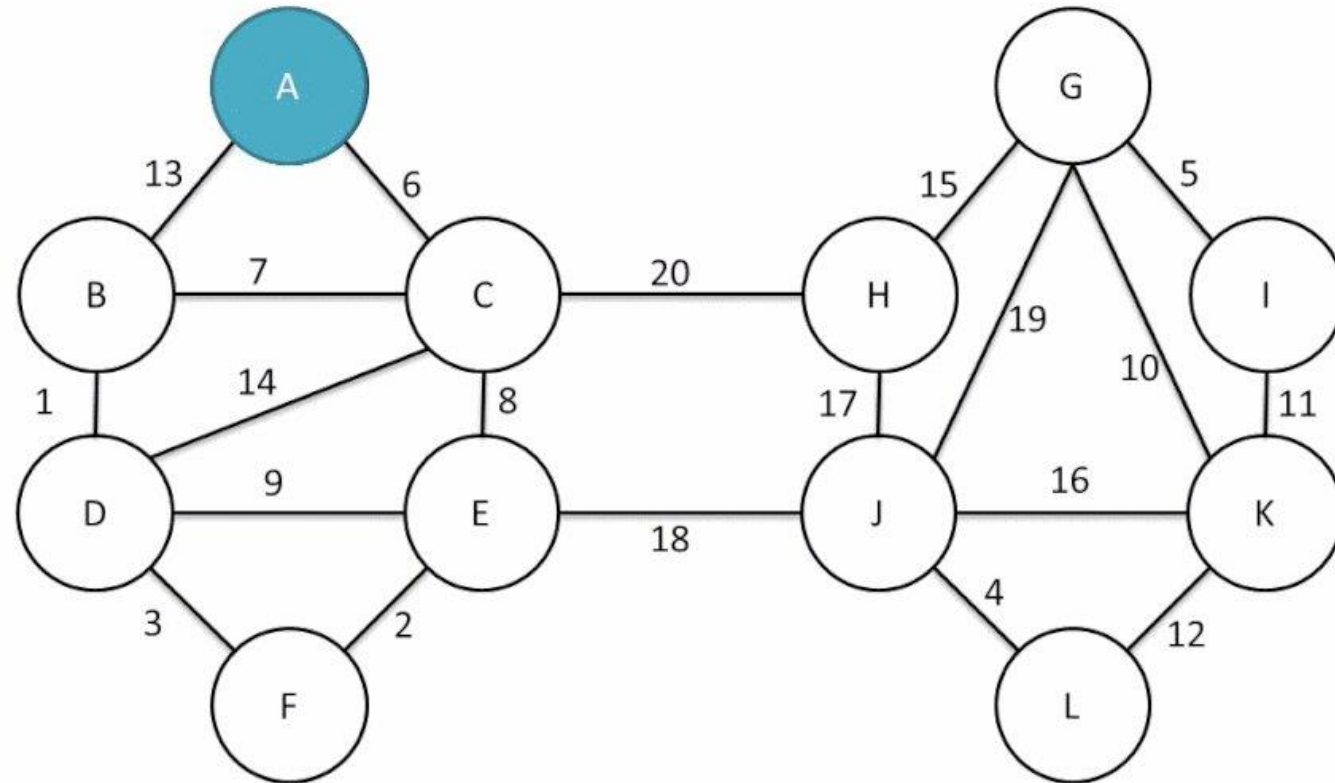


# Borůvka's Algorithm



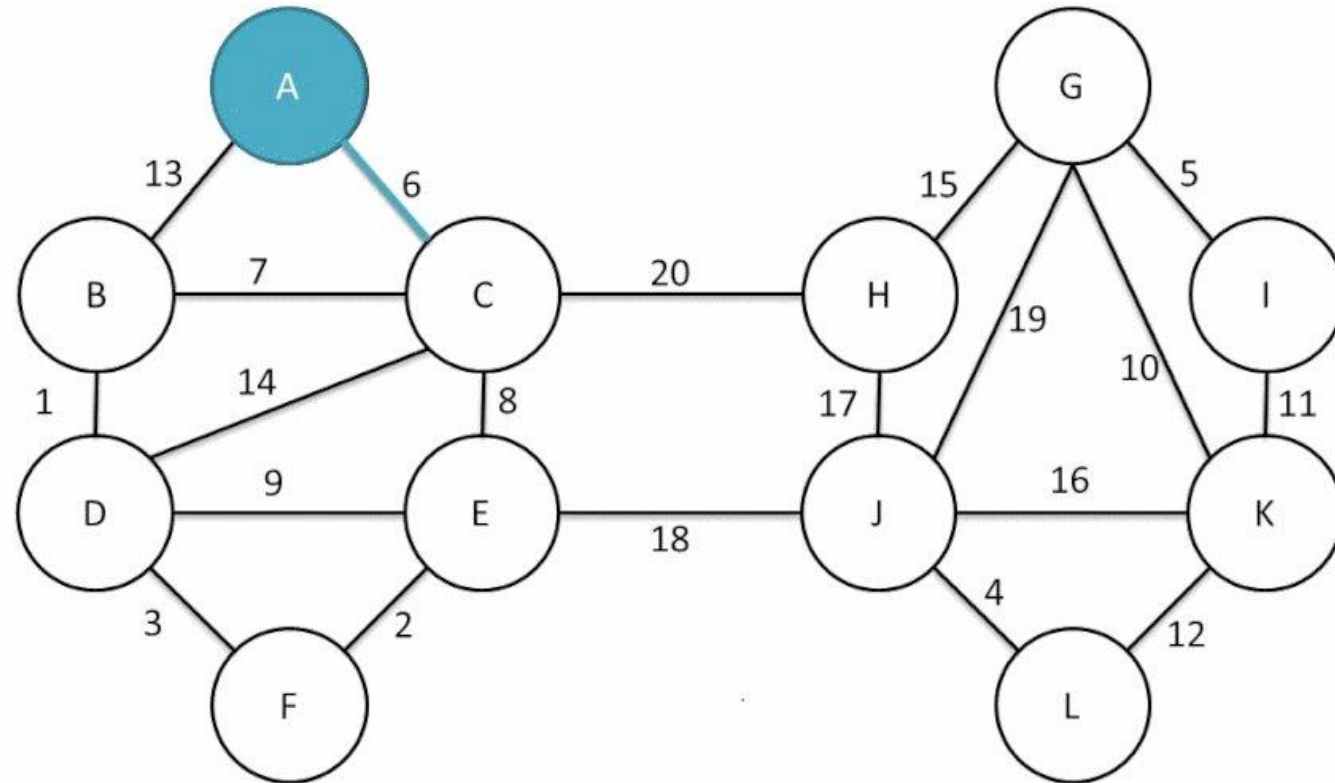


# Borůvka's Algorithm





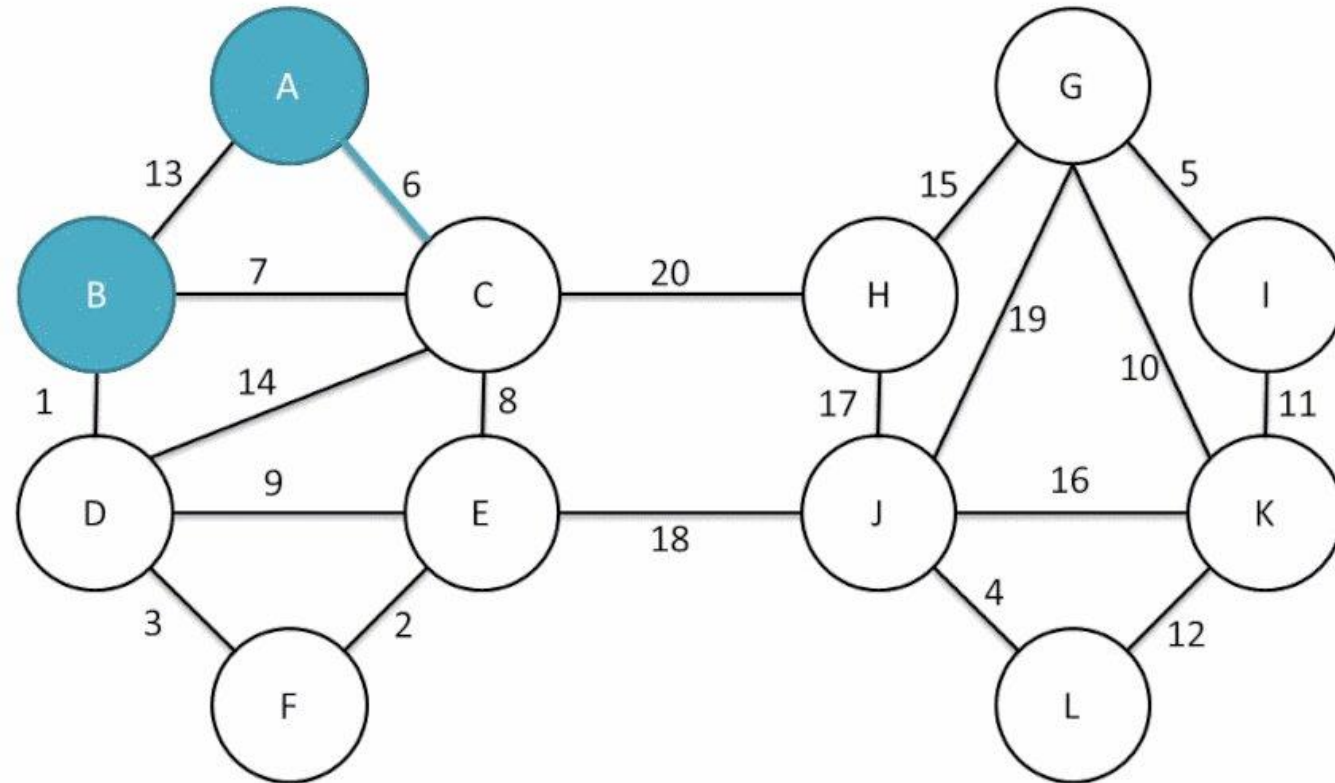
# Borůvka's Algorithm





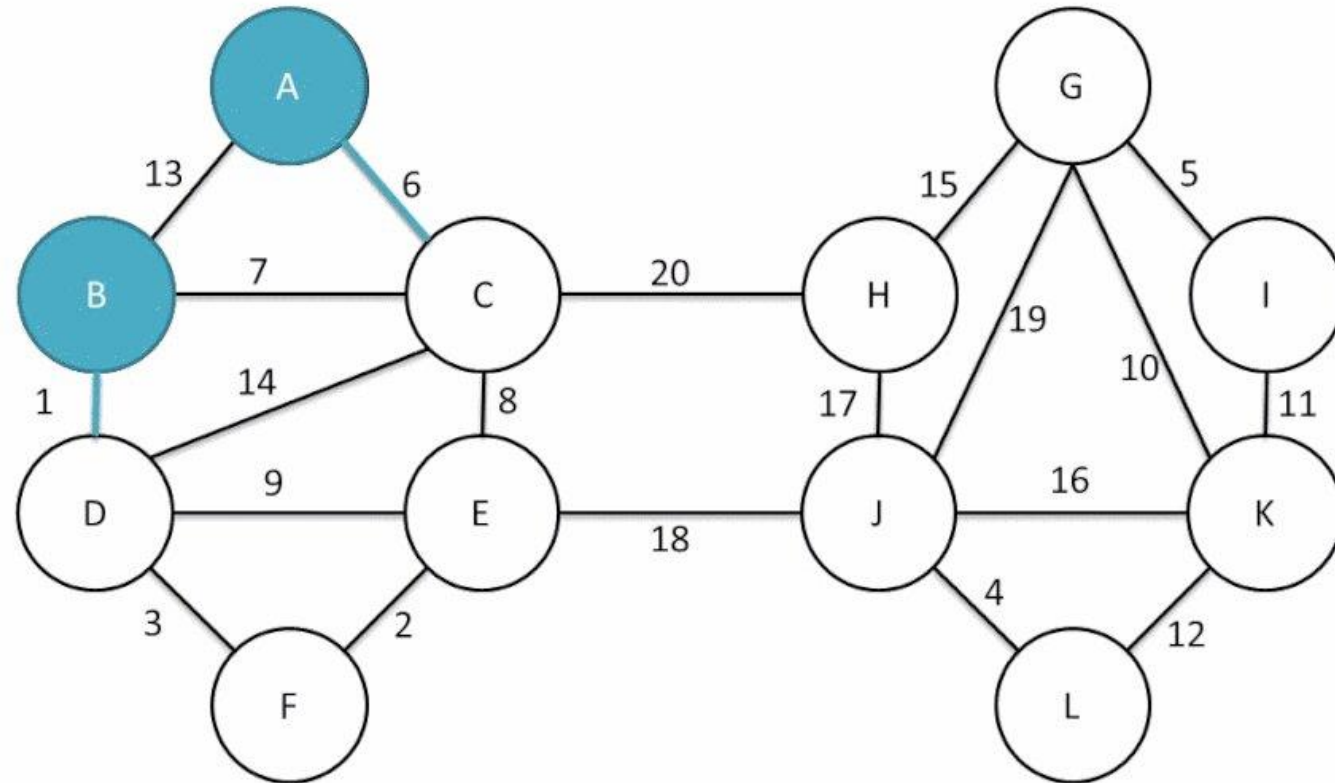


# Borůvka's Algorithm



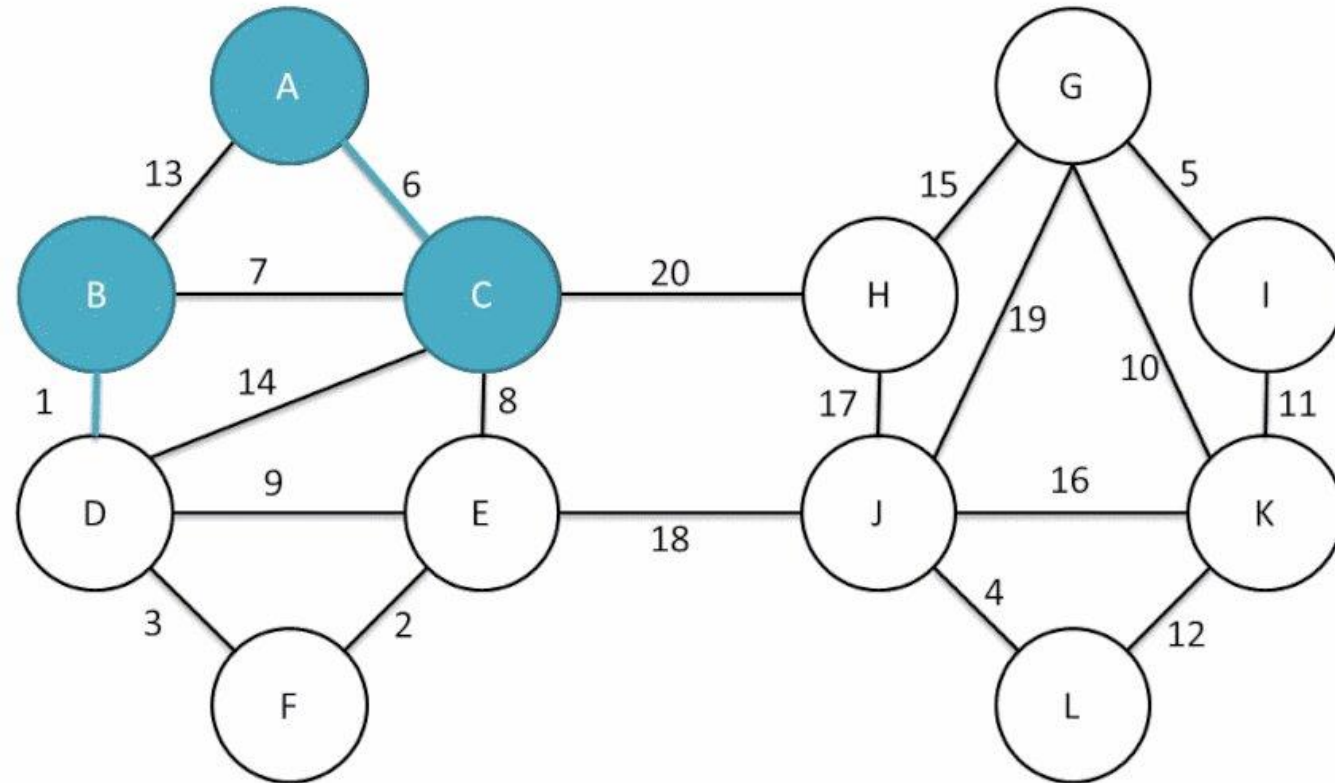


# Borůvka's Algorithm



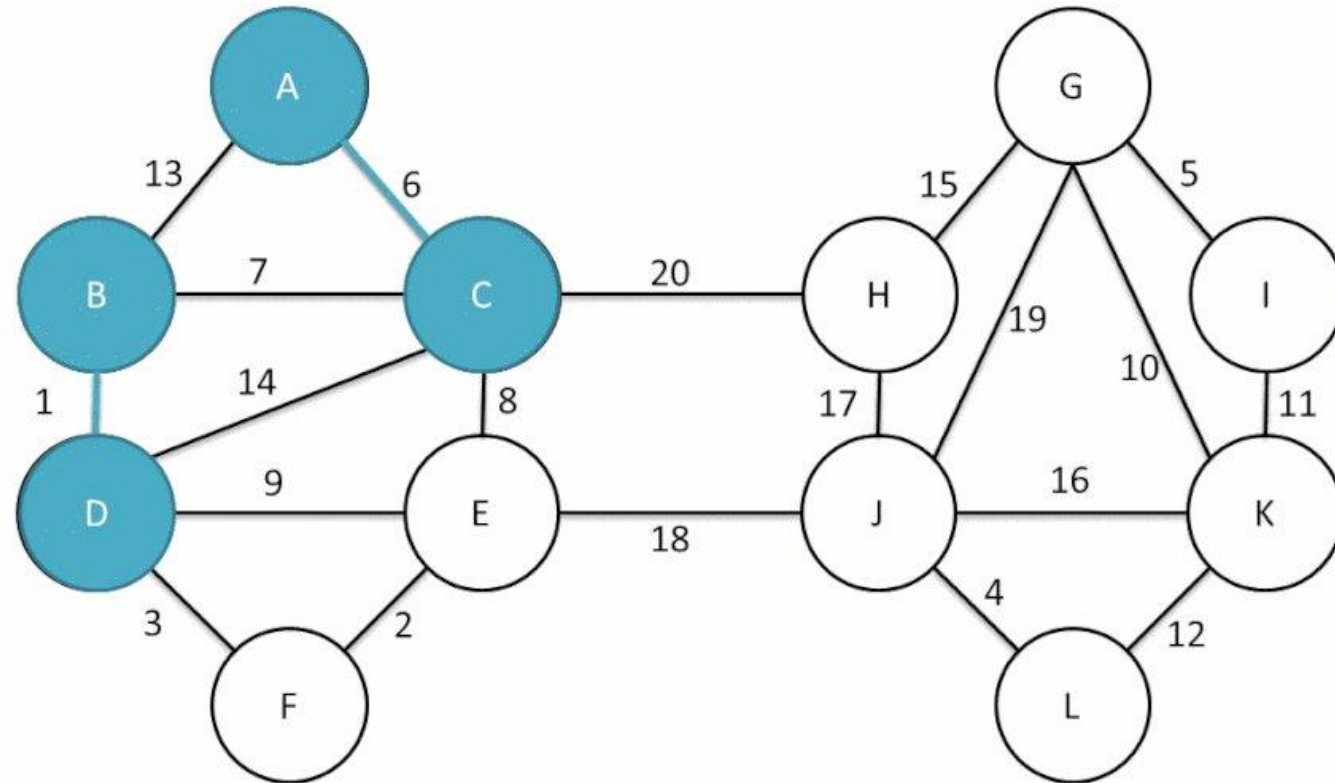


# Borůvka's Algorithm



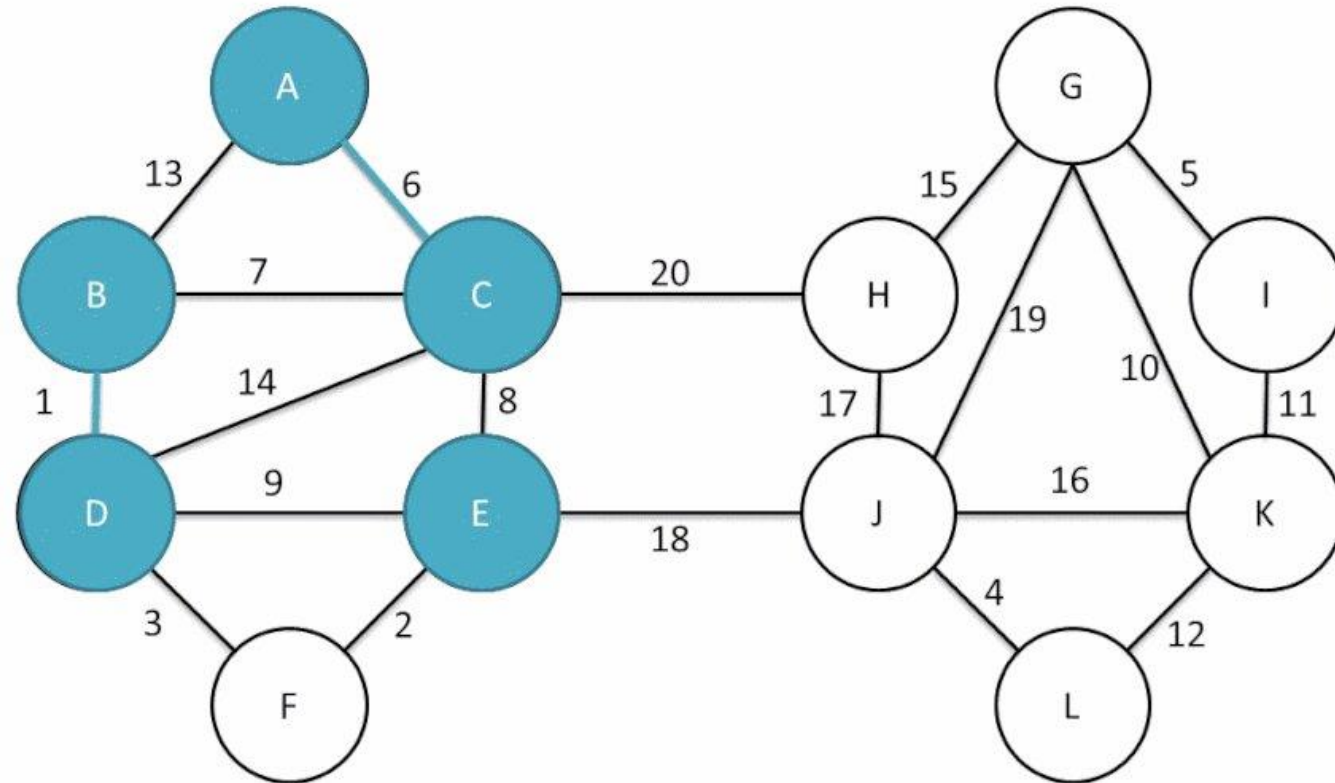


# Borůvka's Algorithm



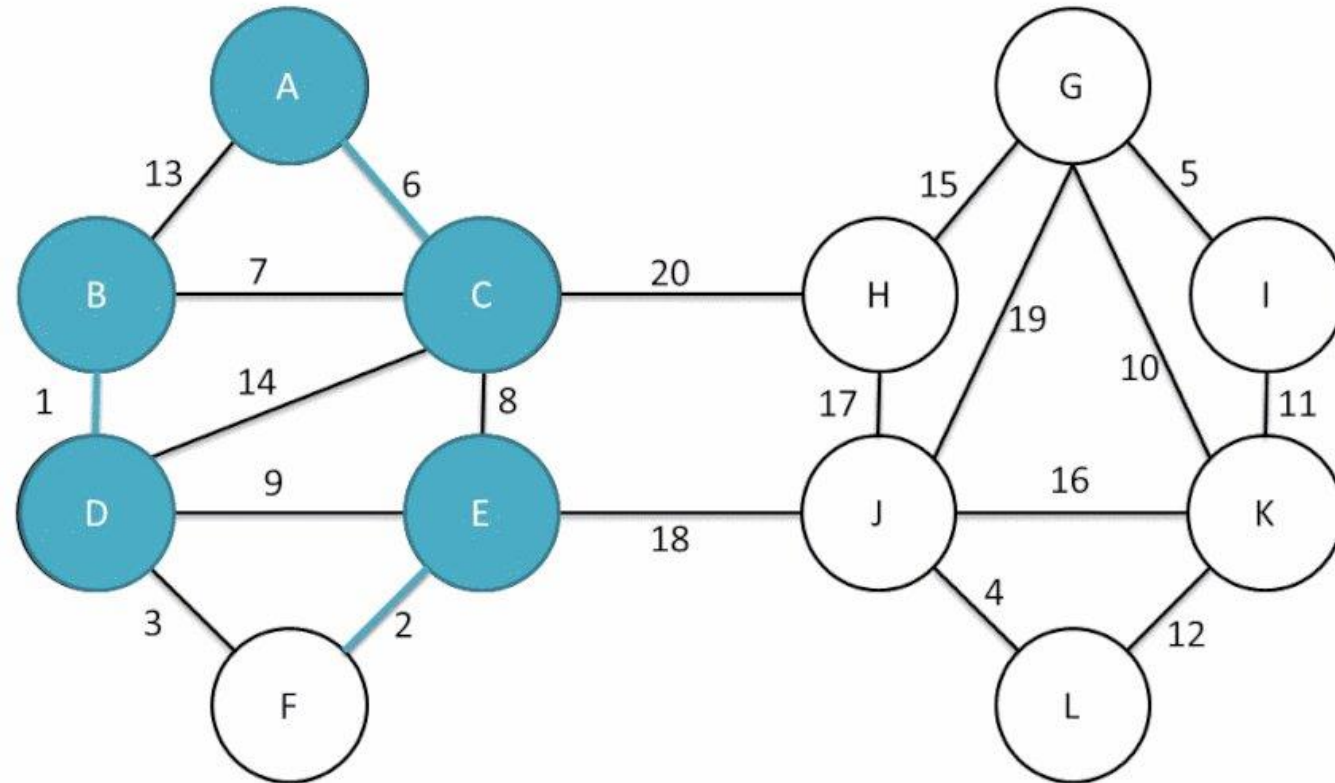


# Borůvka's Algorithm



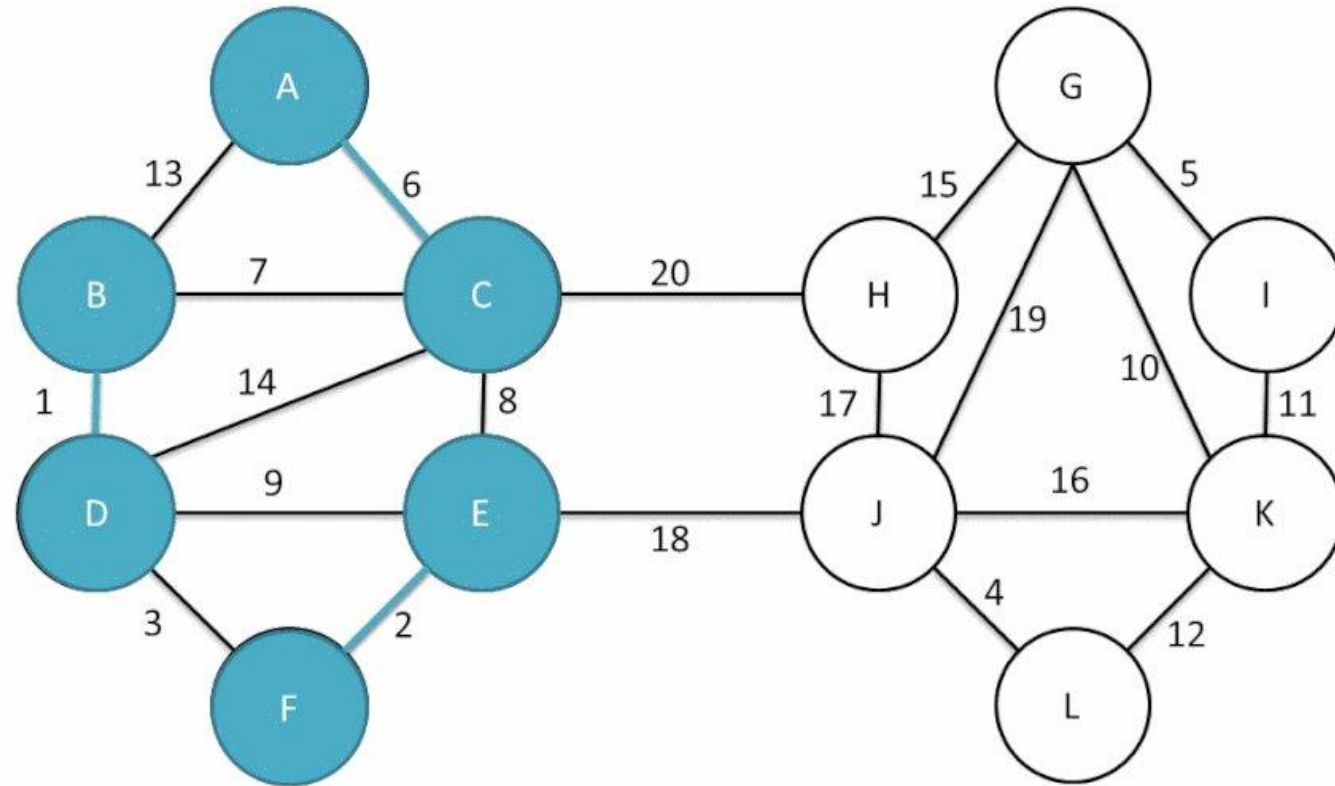


# Borůvka's Algorithm



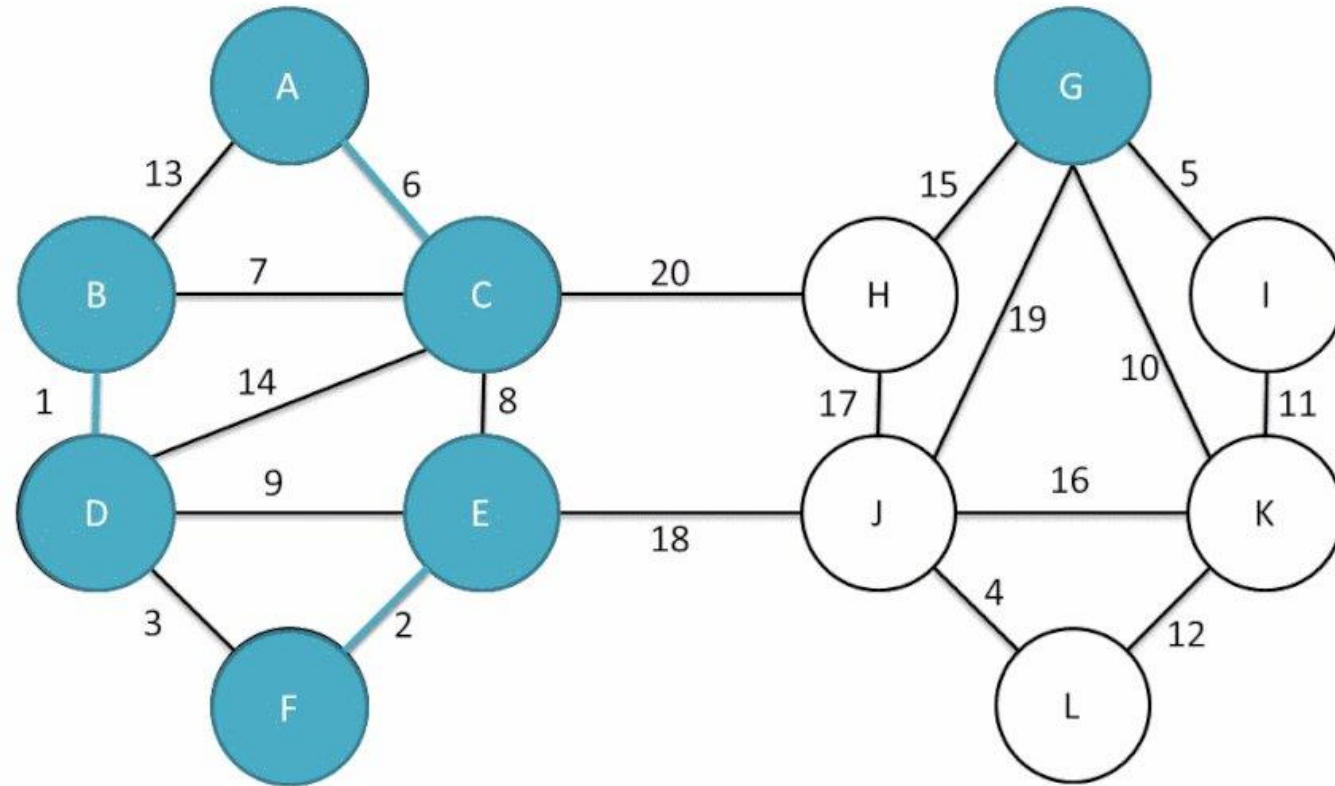


# Borůvka's Algorithm





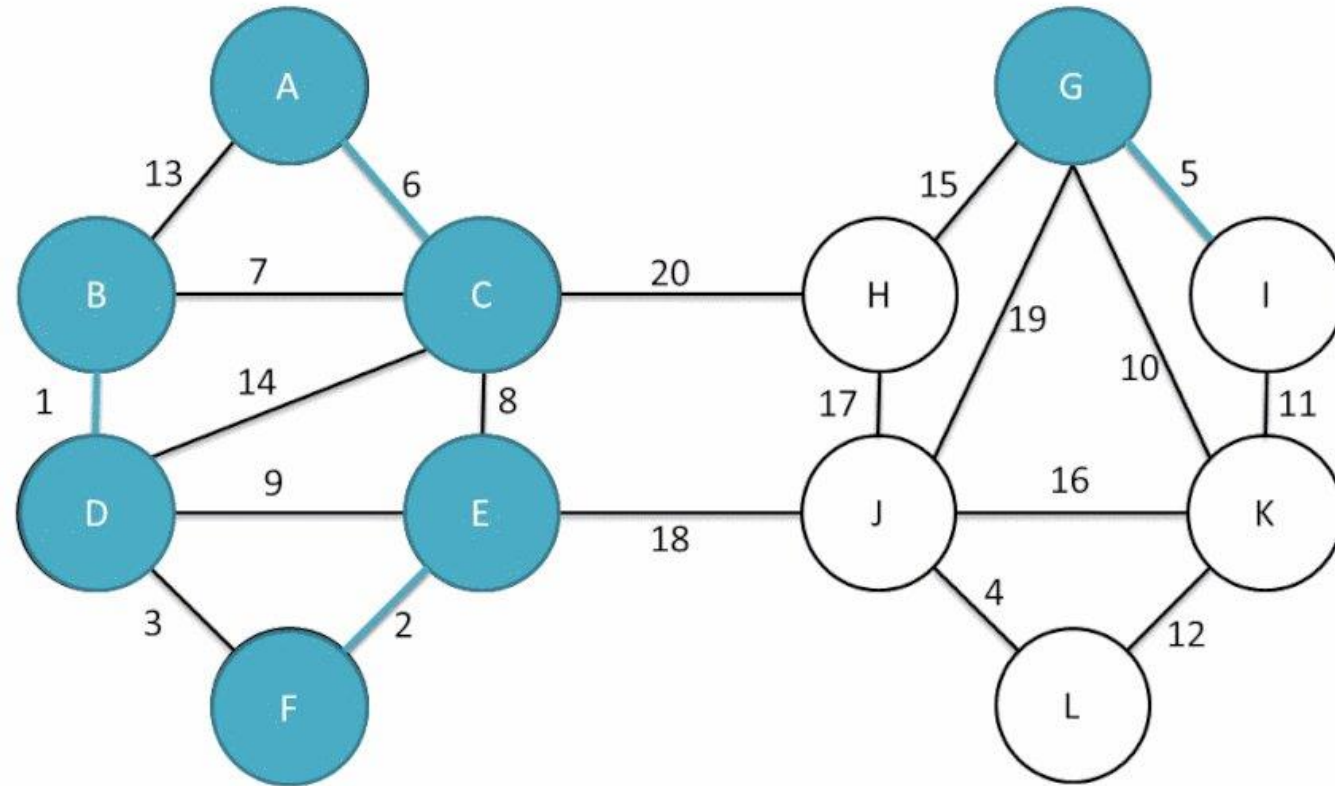
# Borůvka's Algorithm





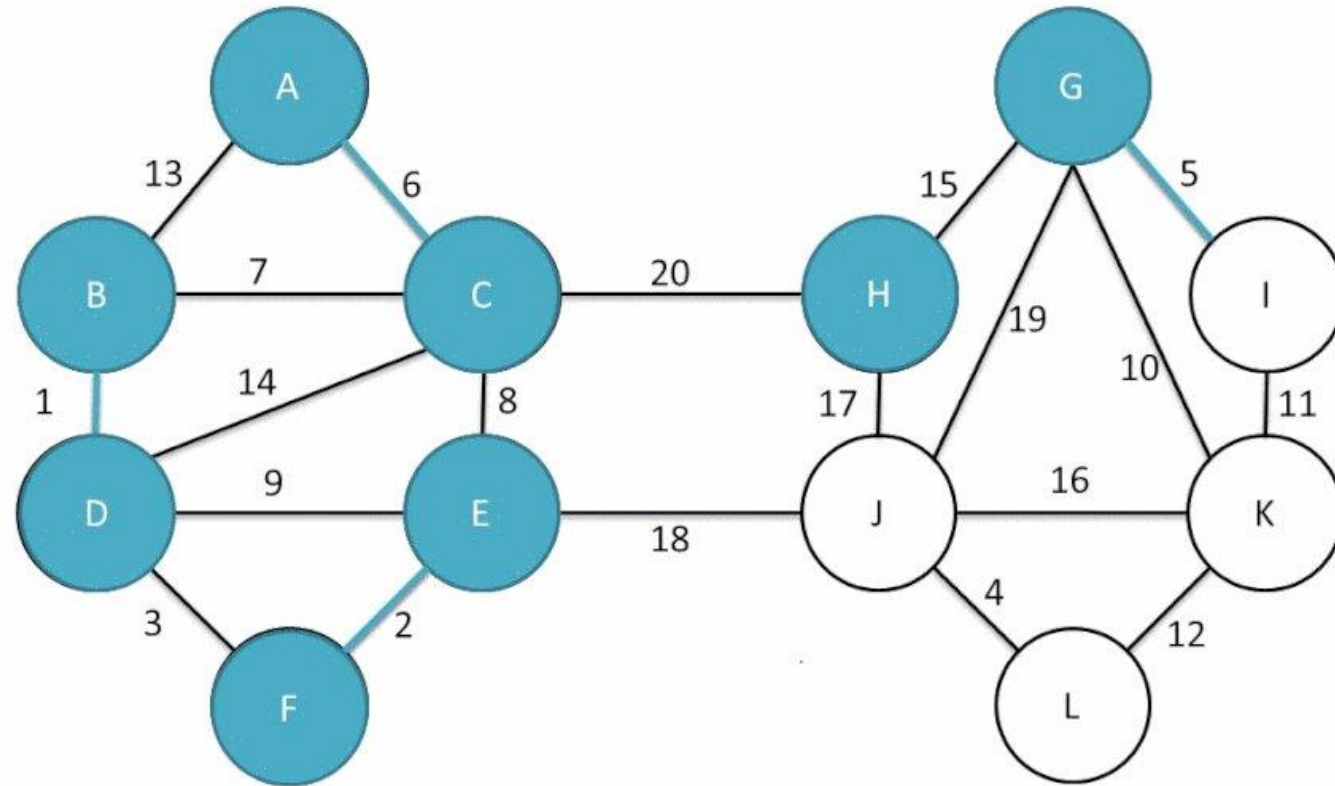


# Borůvka's Algorithm



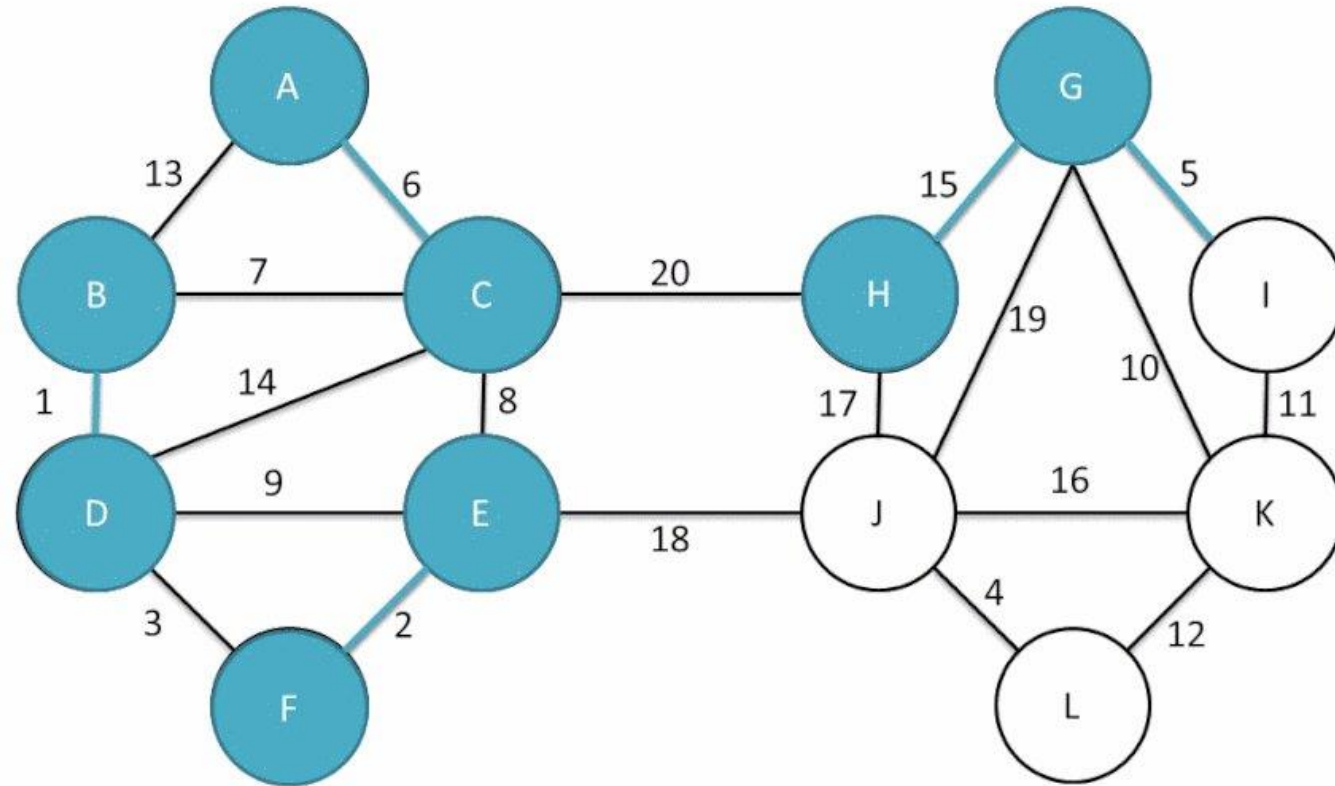


# Borůvka's Algorithm



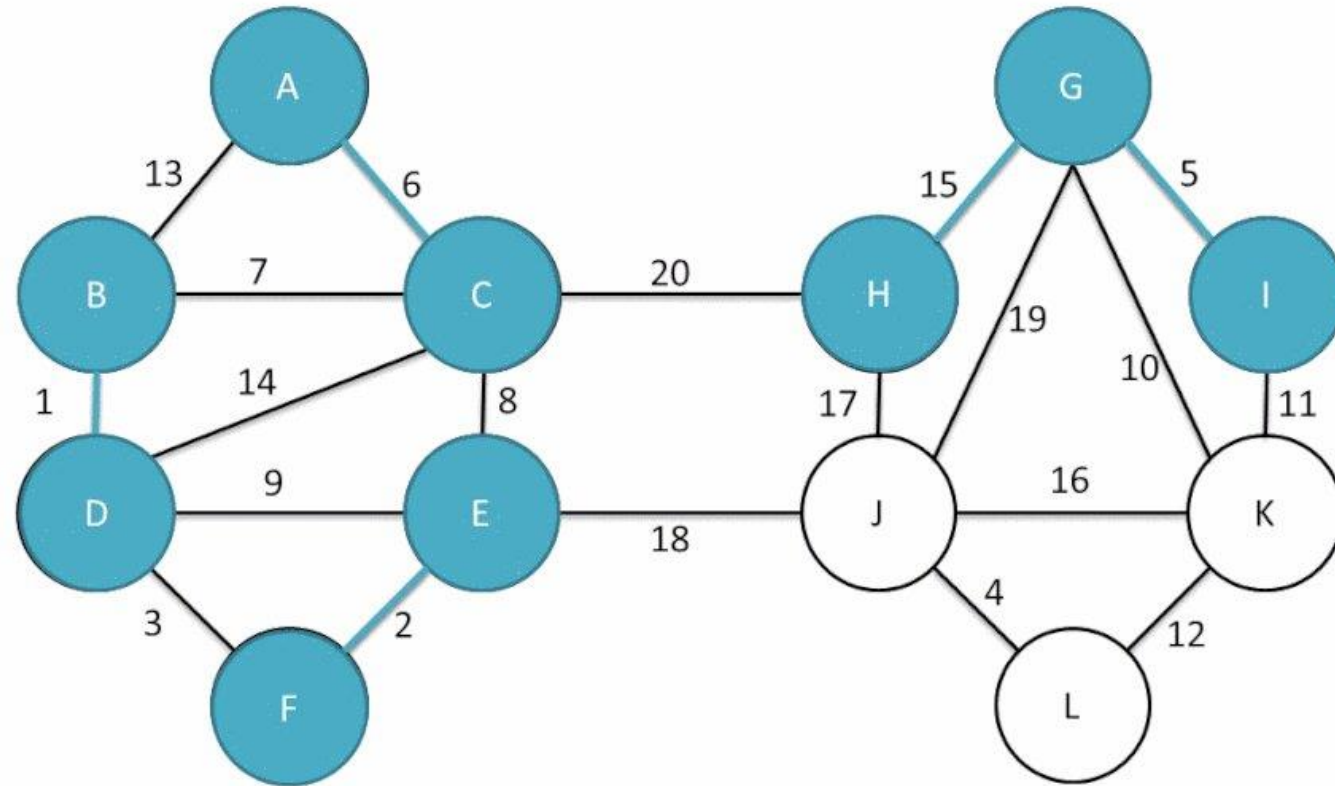


# Borůvka's Algorithm



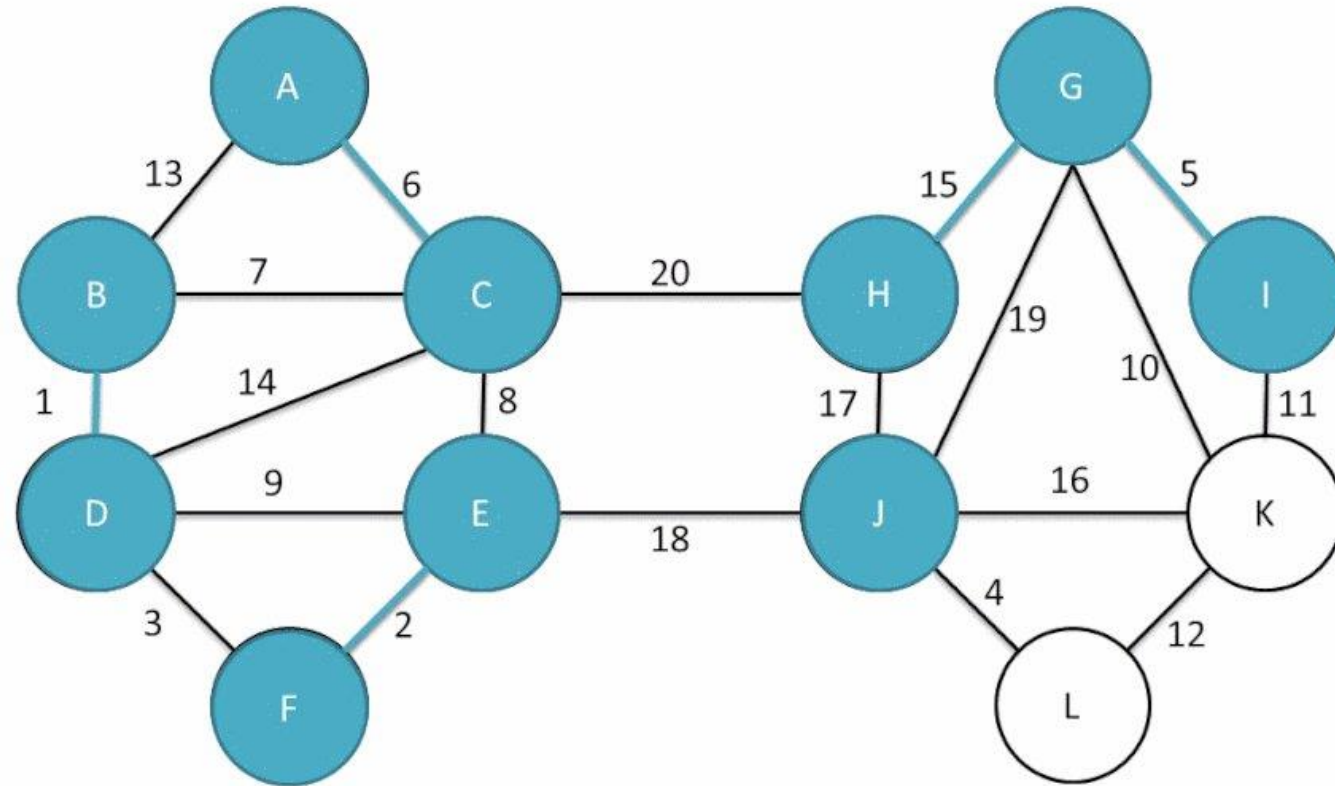


# Borůvka's Algorithm



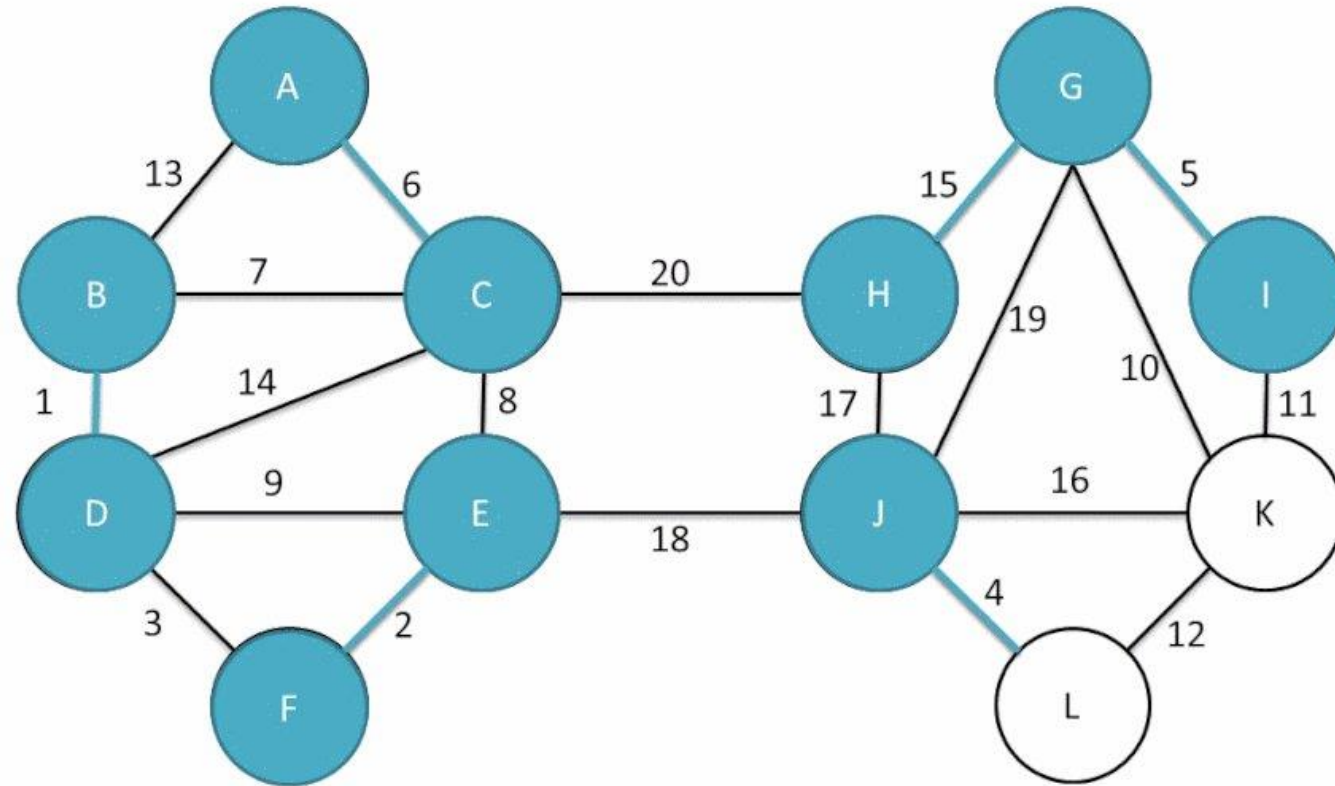


# Borůvka's Algorithm



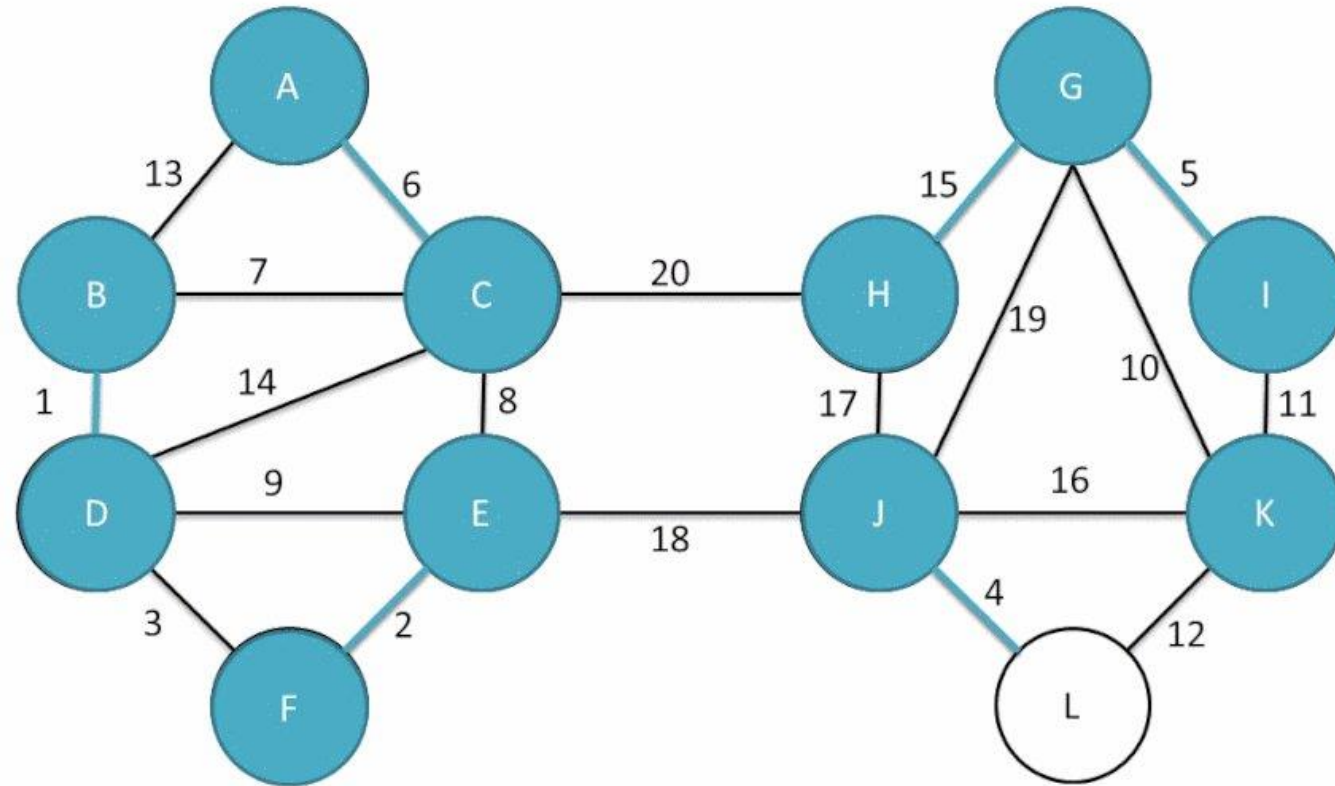


# Borůvka's Algorithm



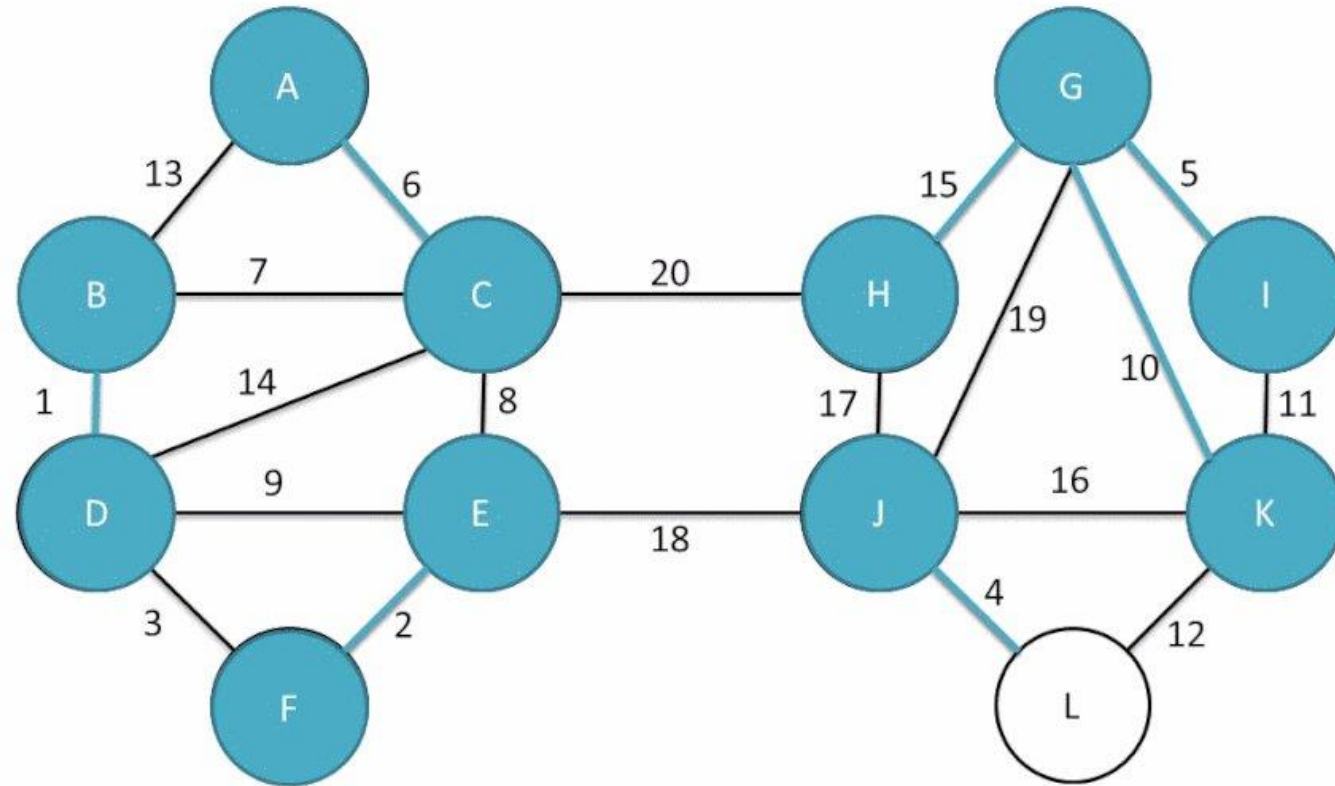


# Borůvka's Algorithm





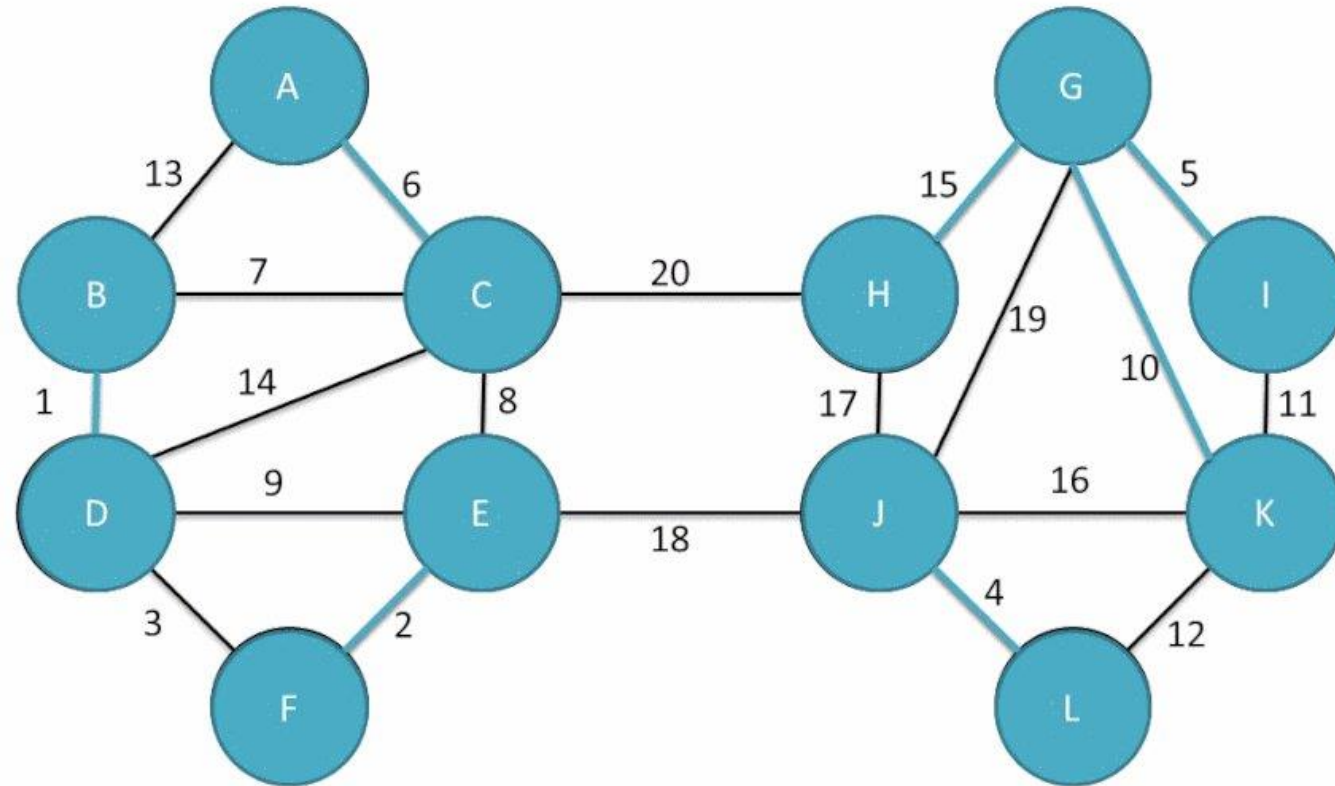
# Borůvka's Algorithm





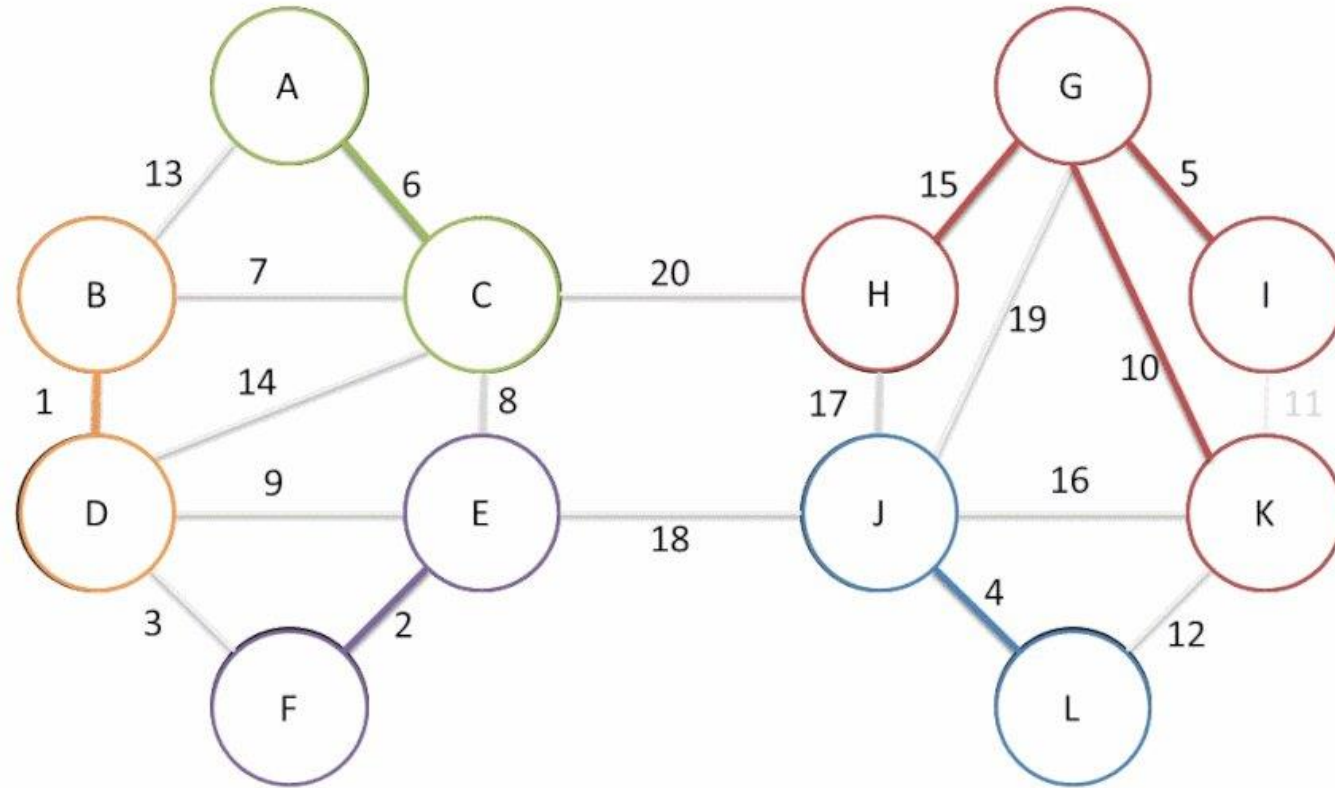


# Borůvka's Algorithm



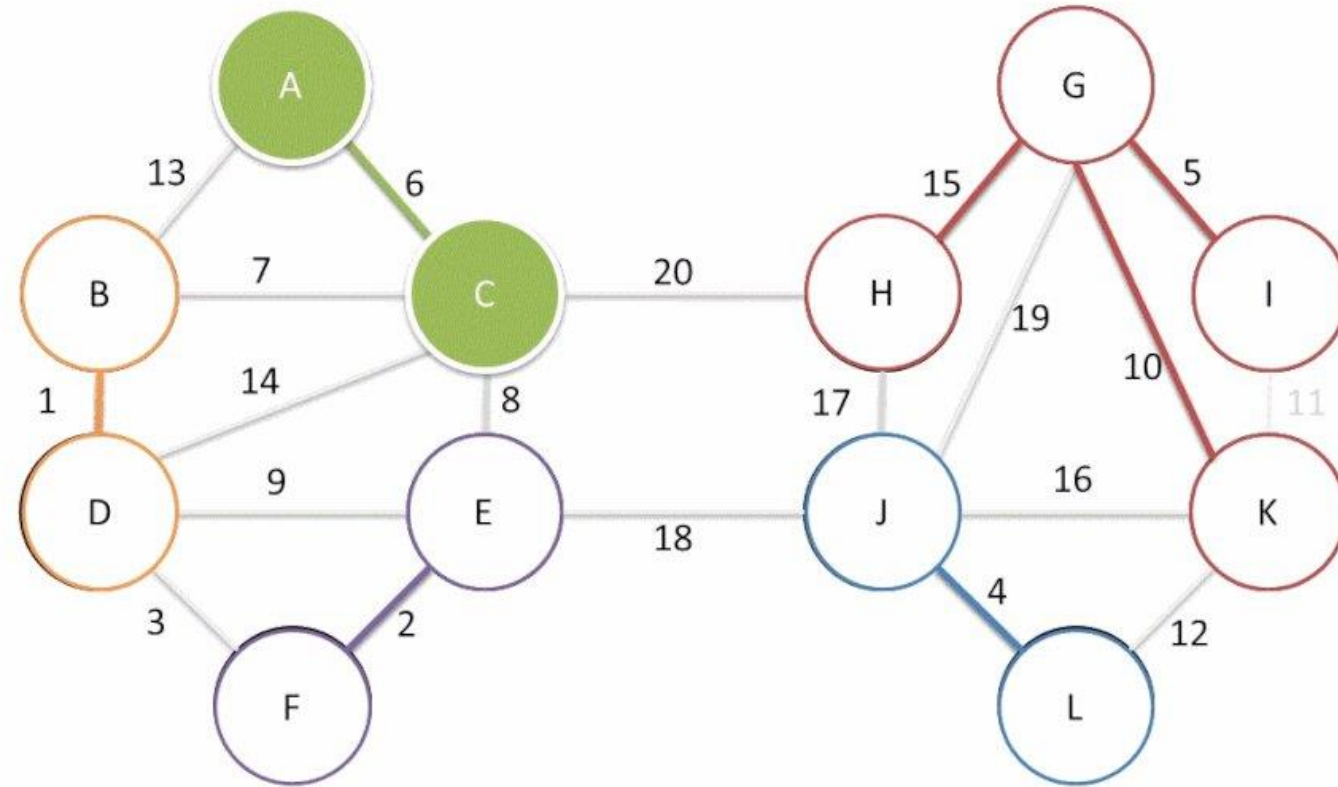


# Borůvka's Algorithm



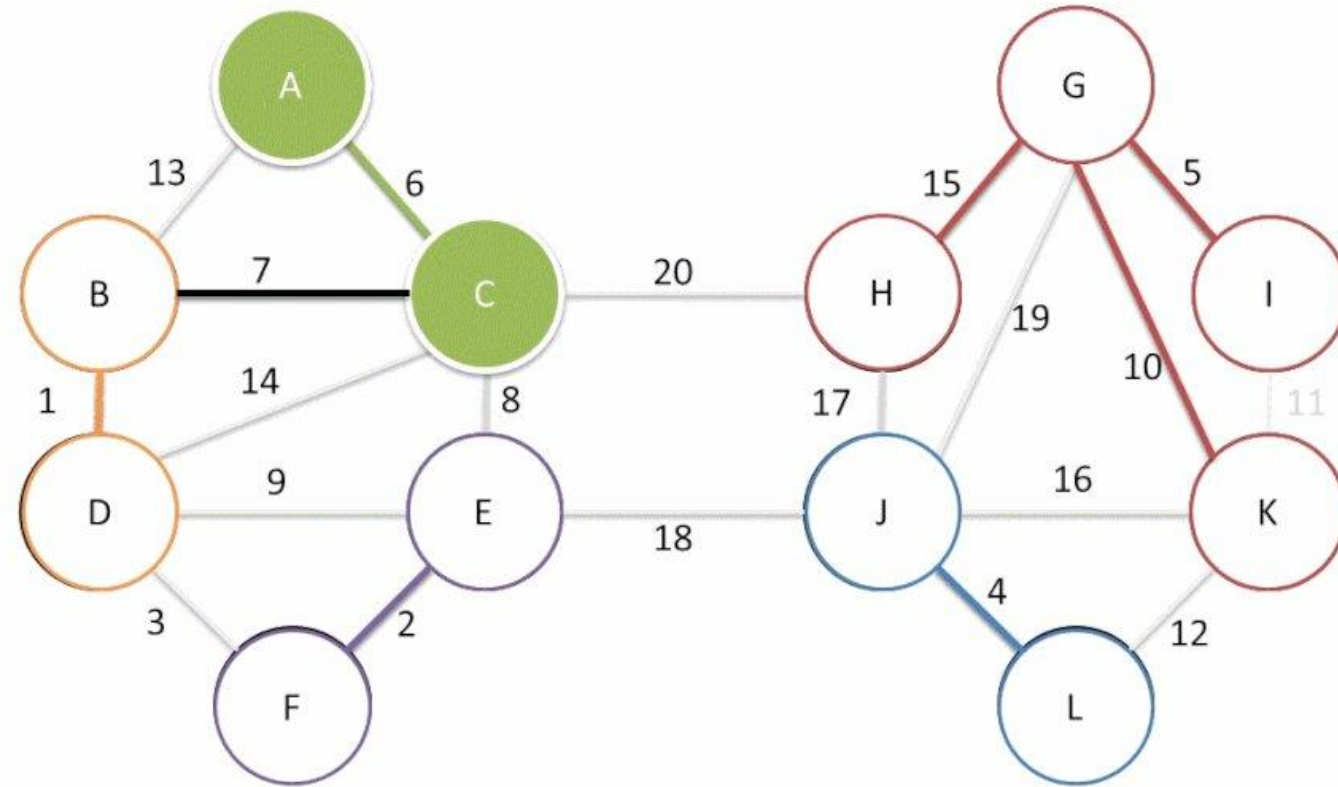


# Borůvka's Algorithm



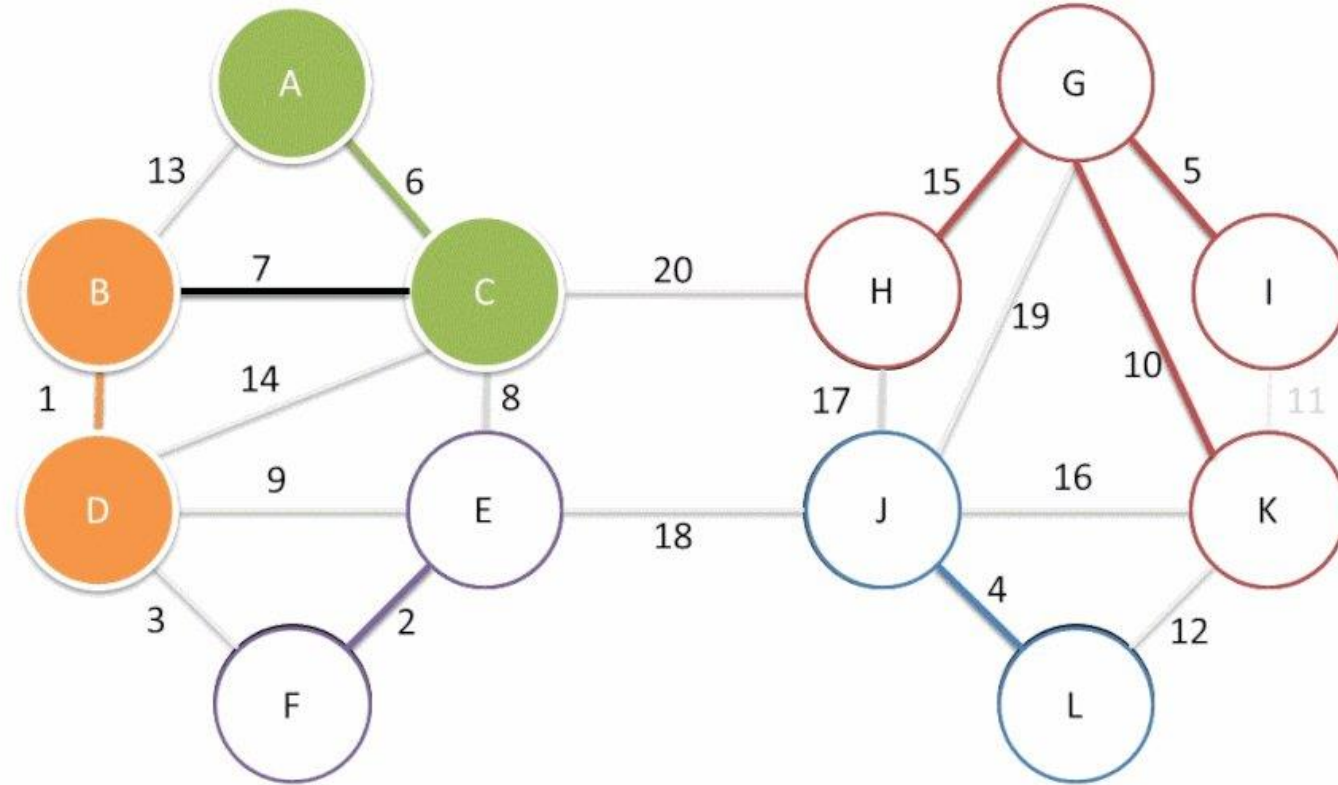


# Borůvka's Algorithm



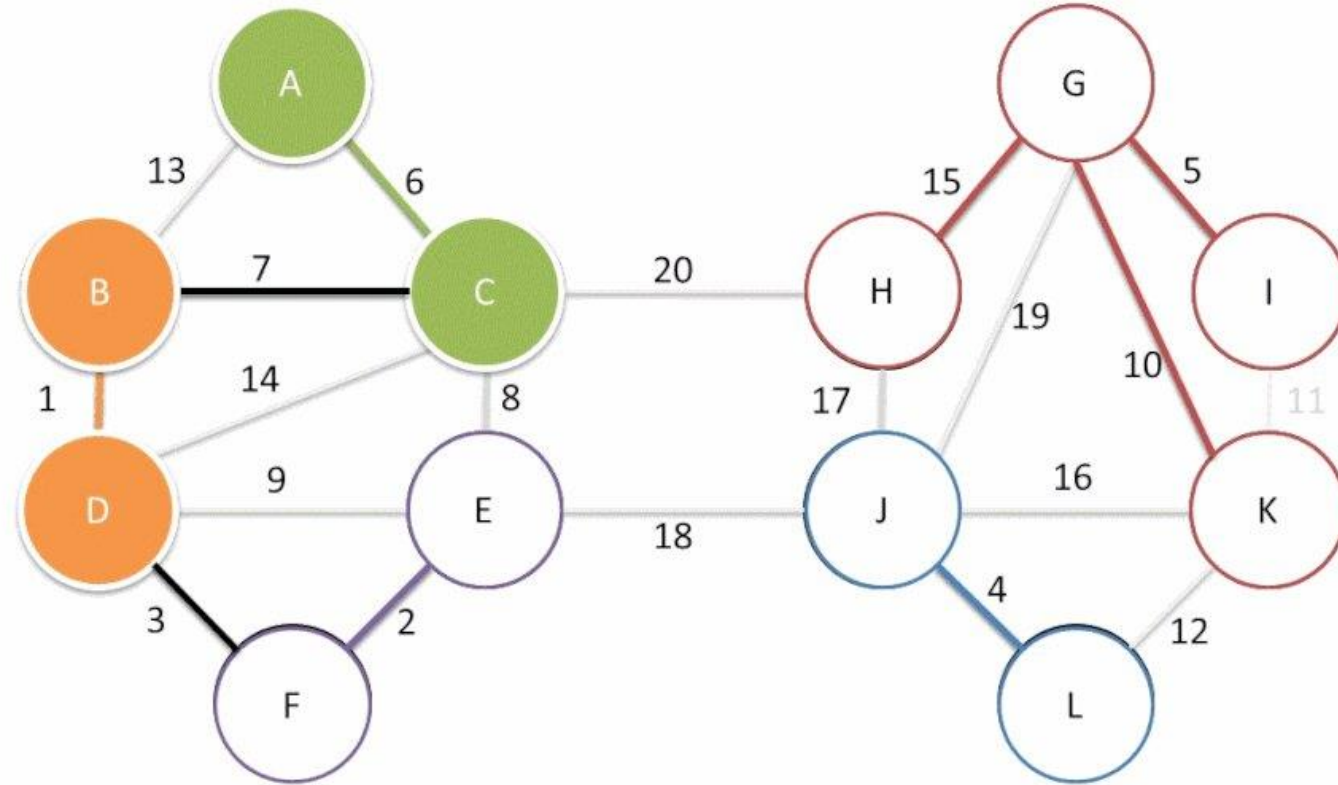


# Borůvka's Algorithm



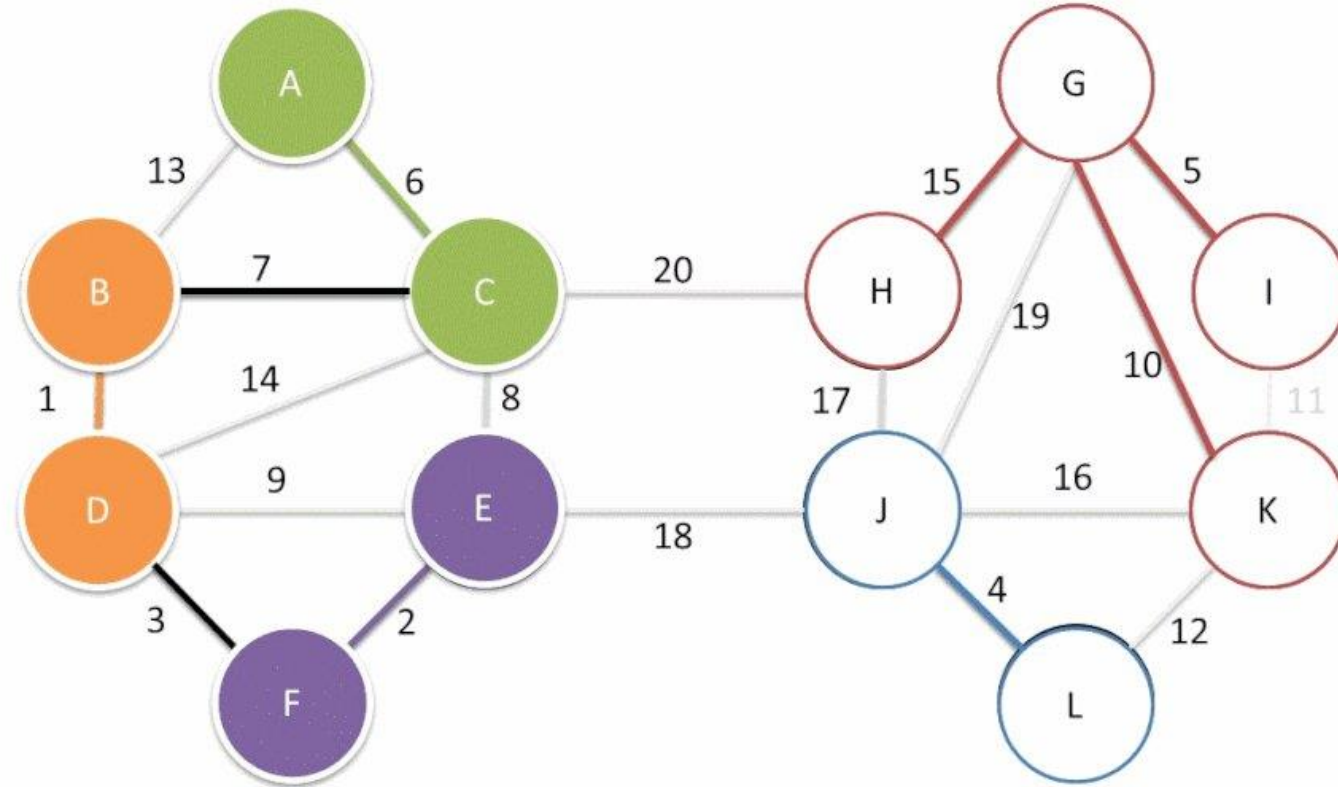


# Borůvka's Algorithm



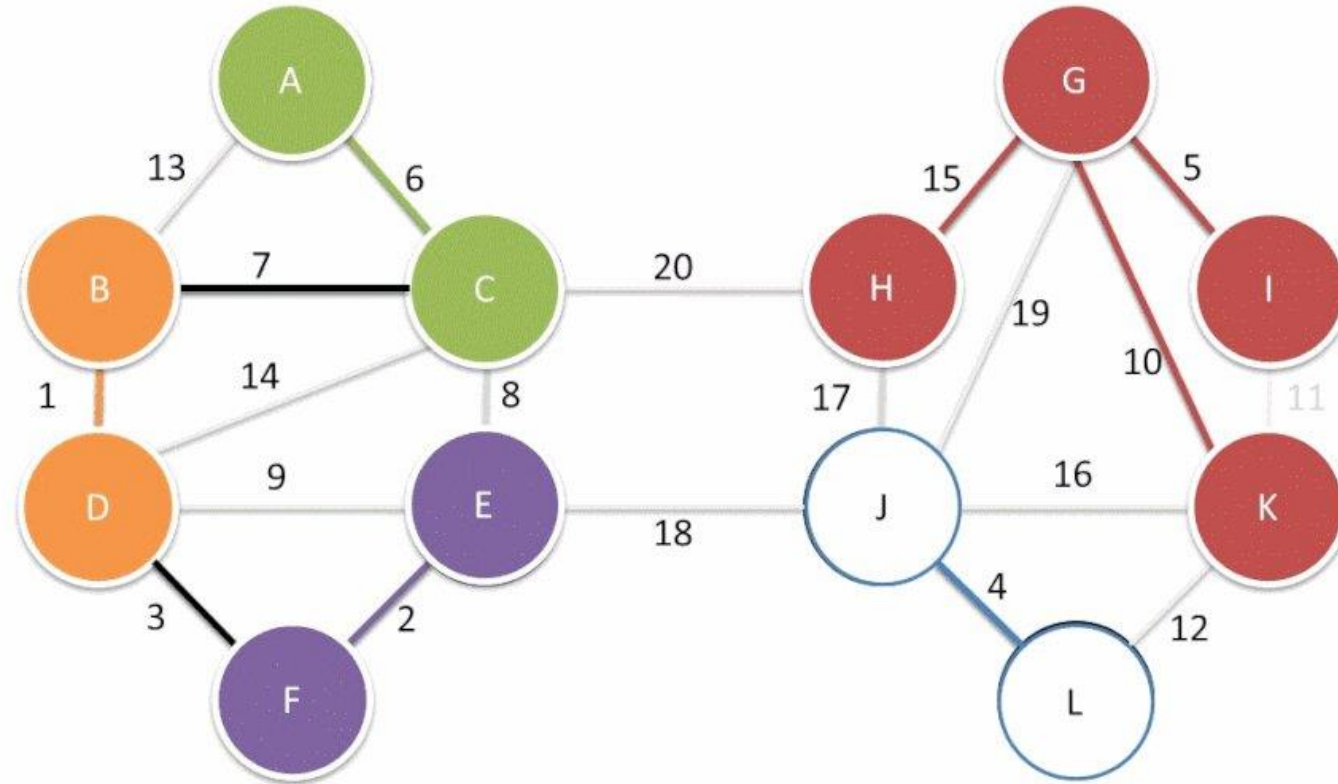


# Borůvka's Algorithm





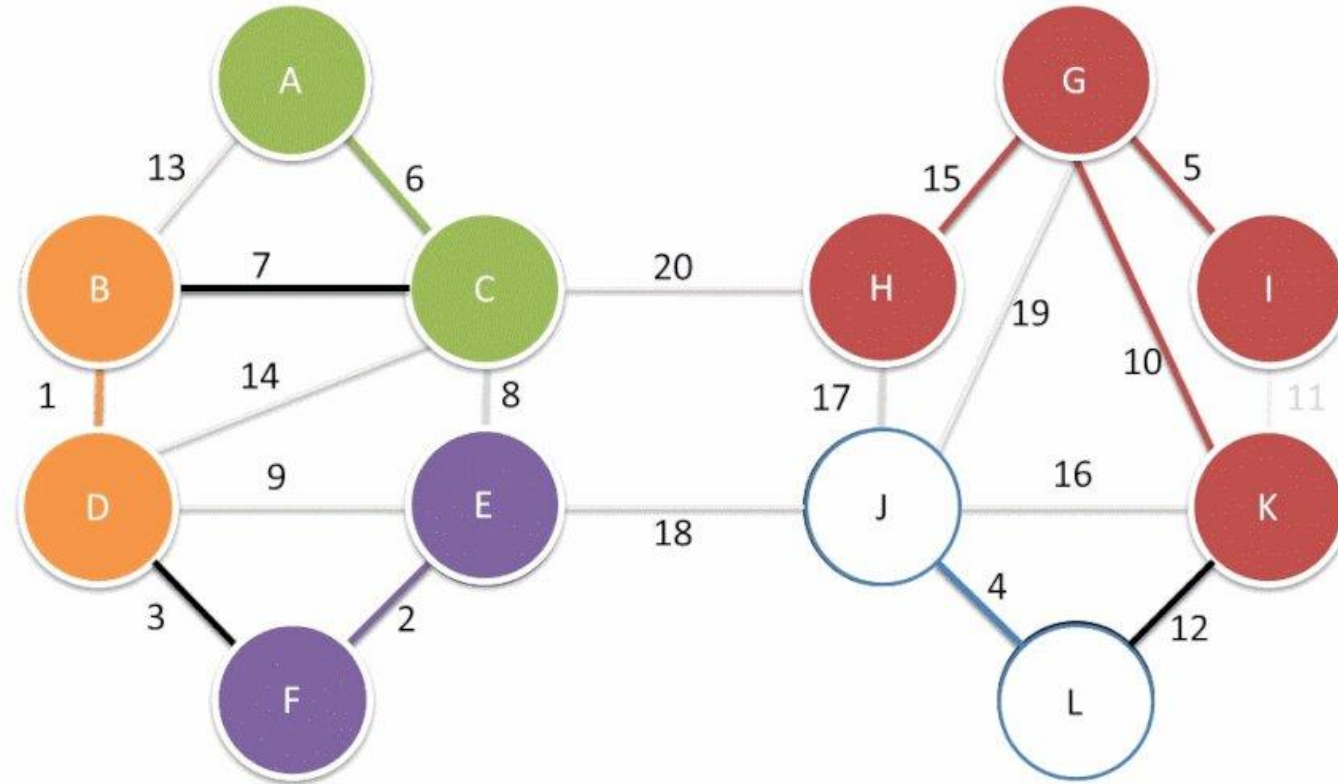
# Borůvka's Algorithm





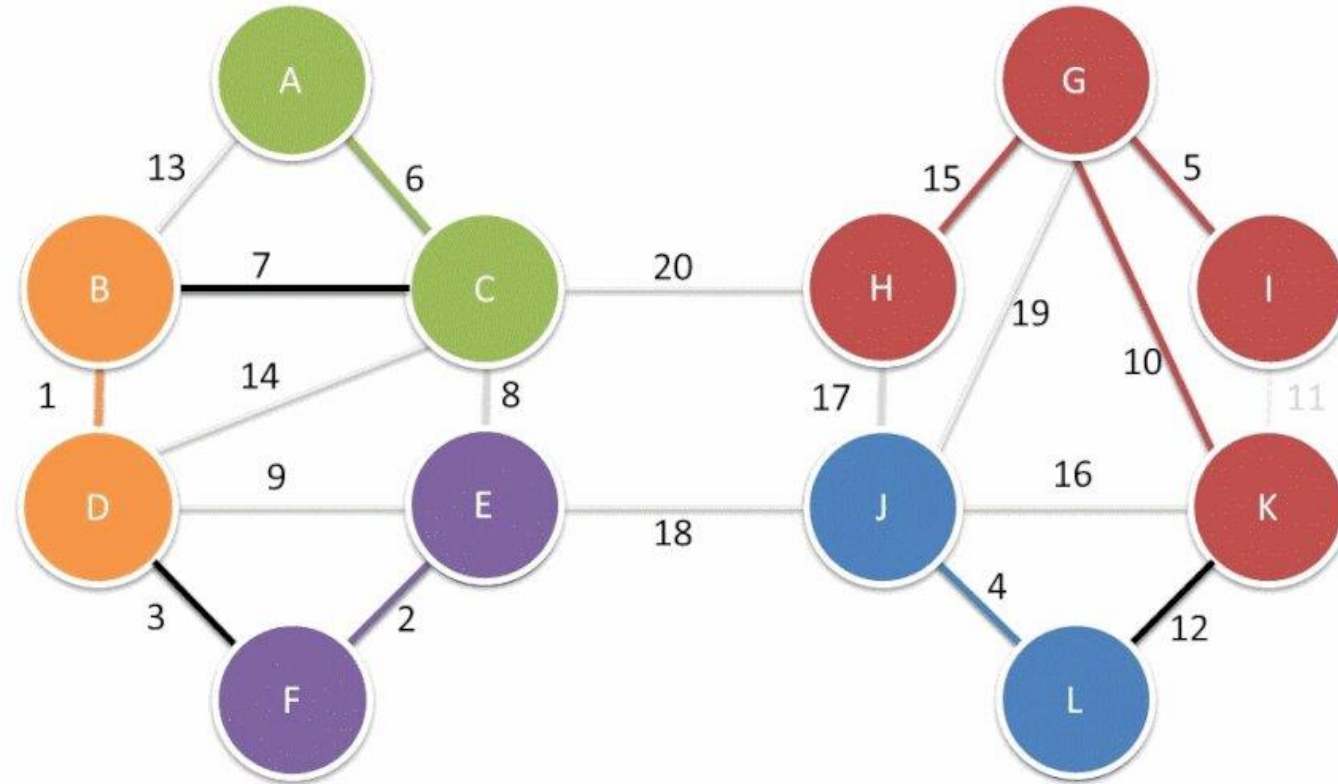


# Borůvka's Algorithm



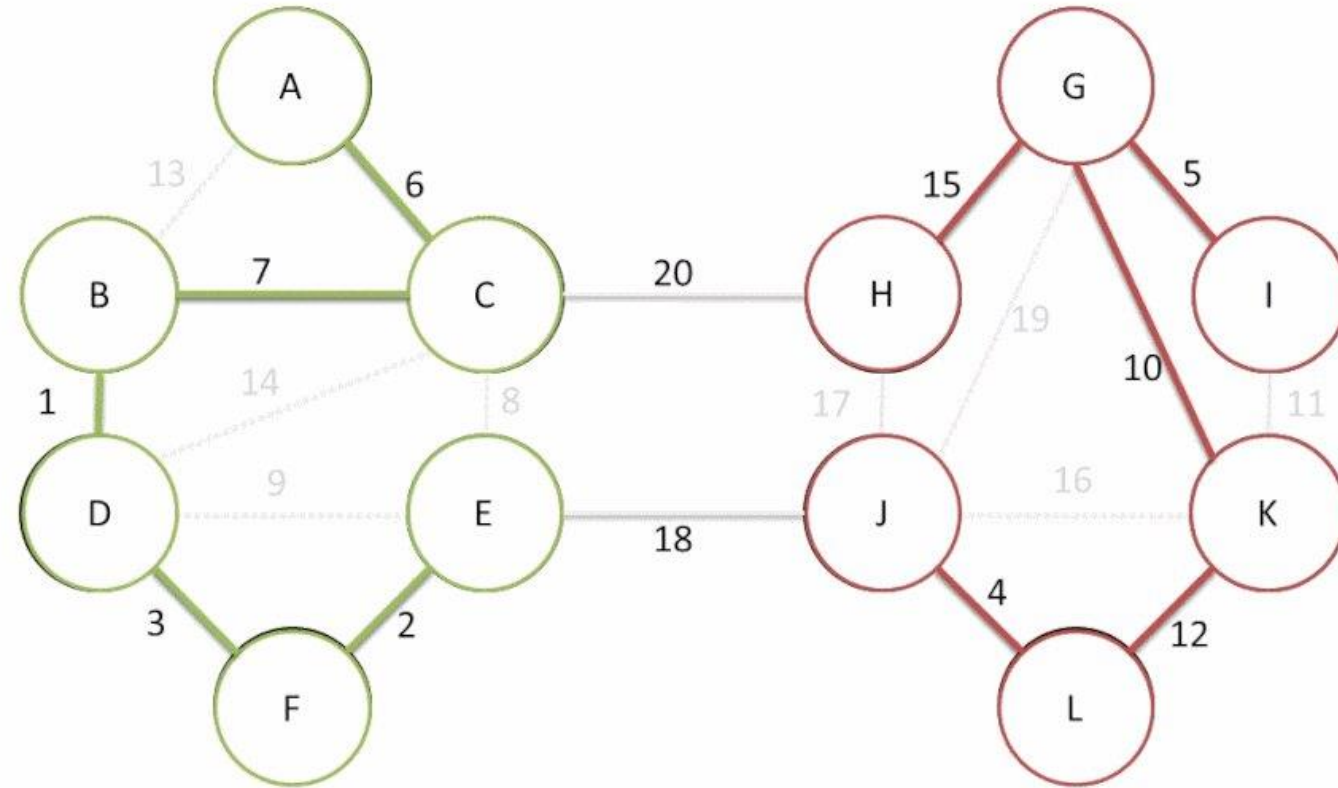


# Borůvka's Algorithm



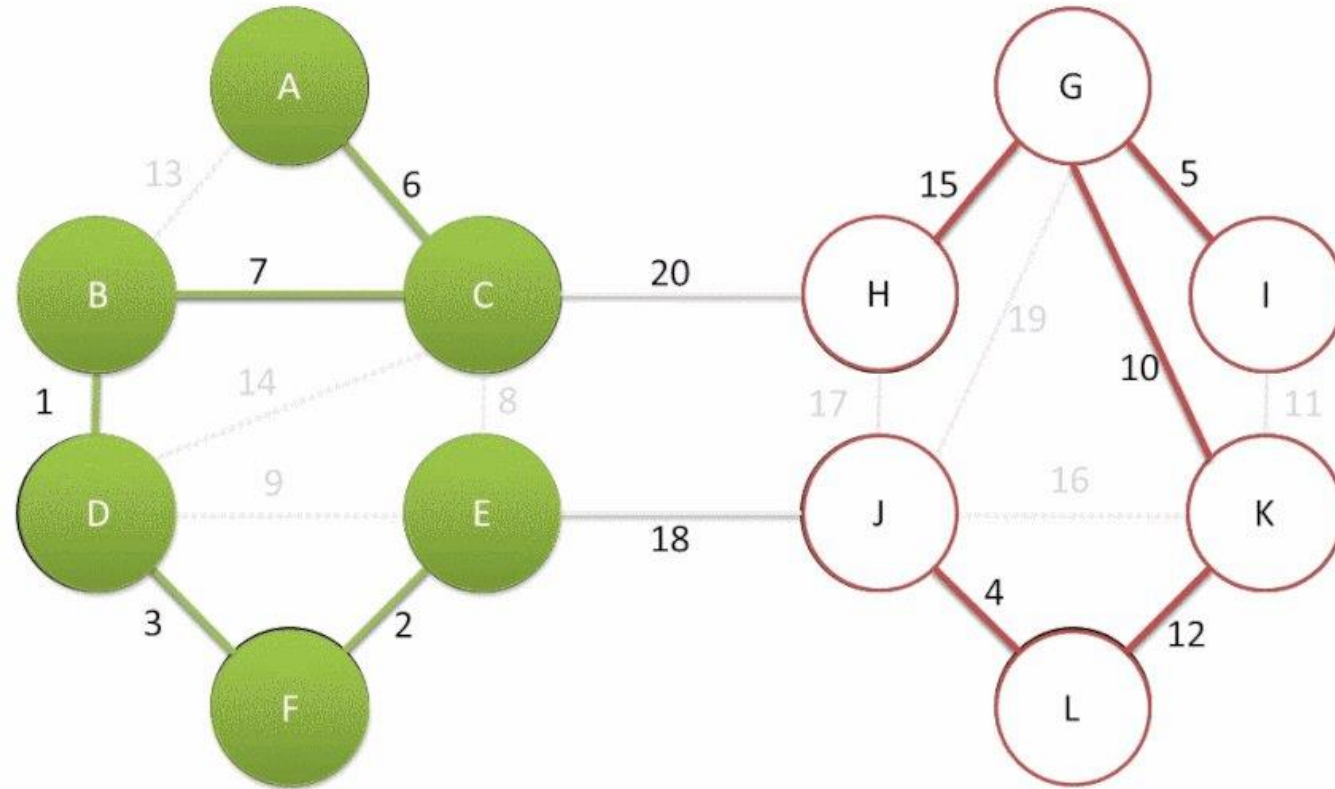


# Borůvka's Algorithm



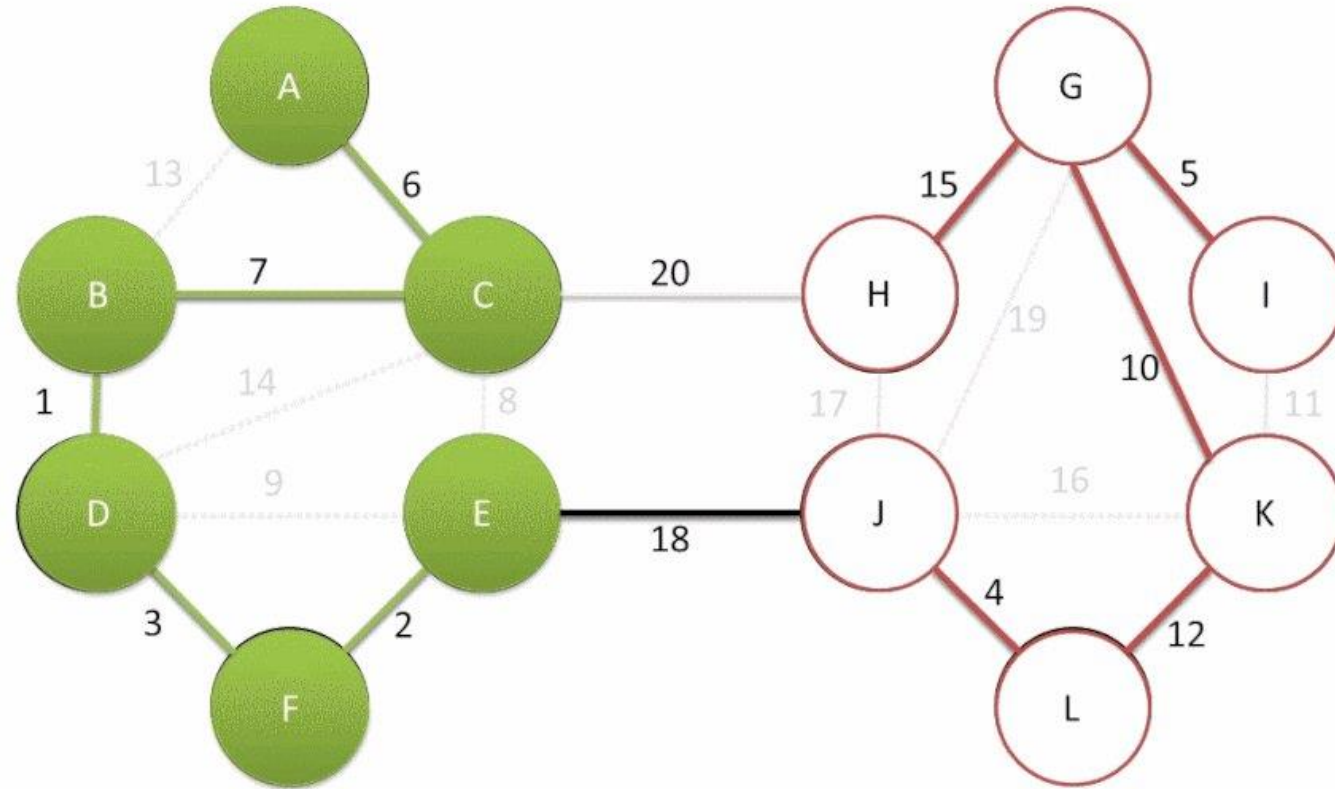


# Borůvka's Algorithm



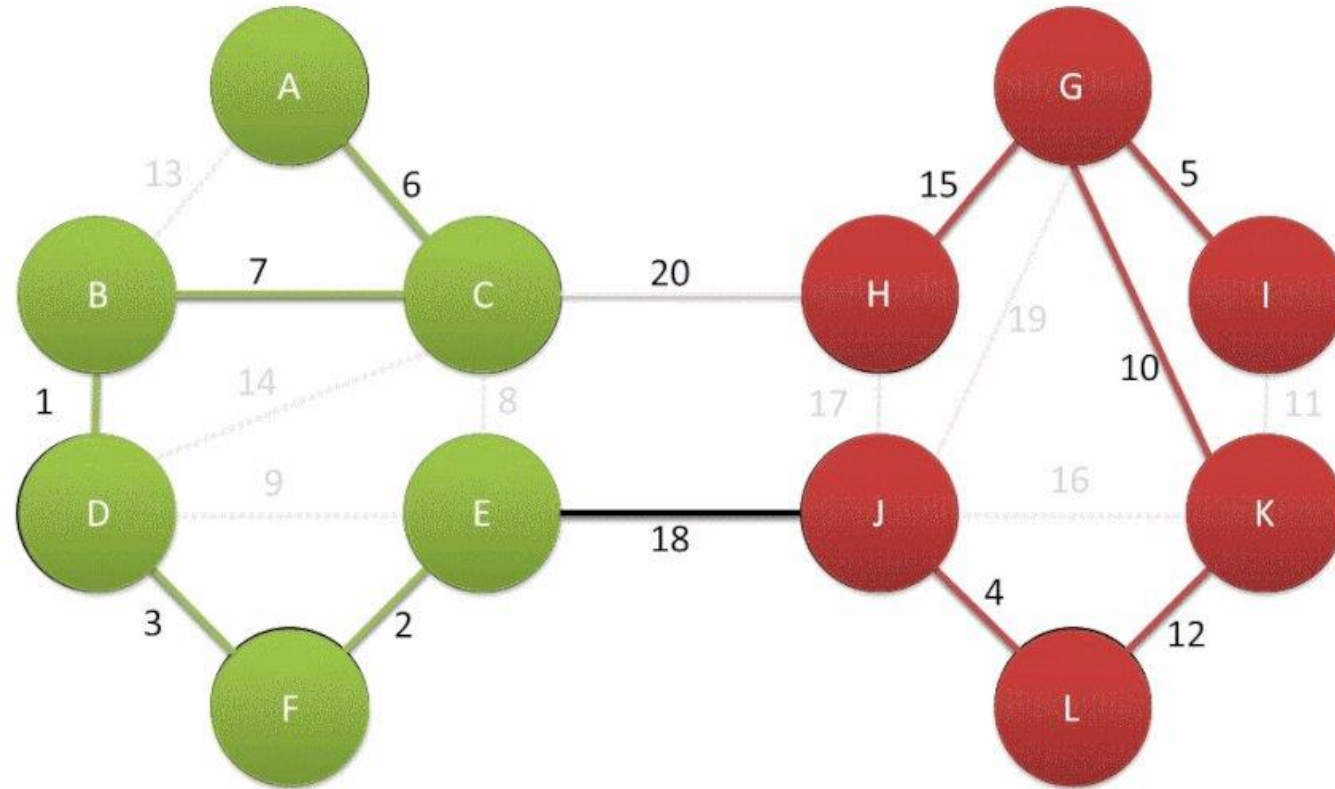


# Borůvka's Algorithm



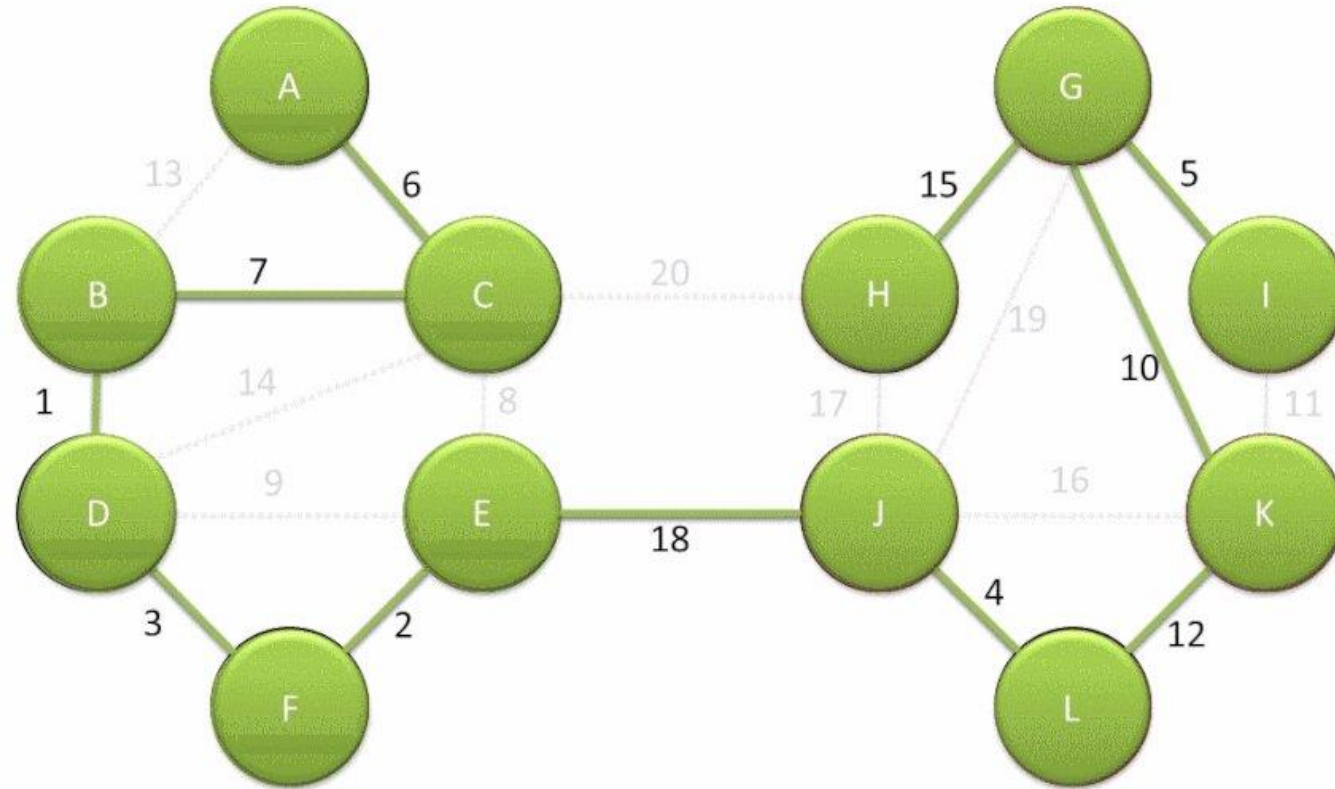


# Borůvka's Algorithm





# Borůvka's Algorithm









# Huffman Kodlama

- Veri sıkıştırma için kullanılır.
- Değişken uzunluklu kodlar kullanır.
- Daha sık kullanılan karakterlere daha kısa kodlar atanır.
- Prefix-free kodlar üretir:
  - Hiçbir kod, başka bir kodun ön eki olamaz.

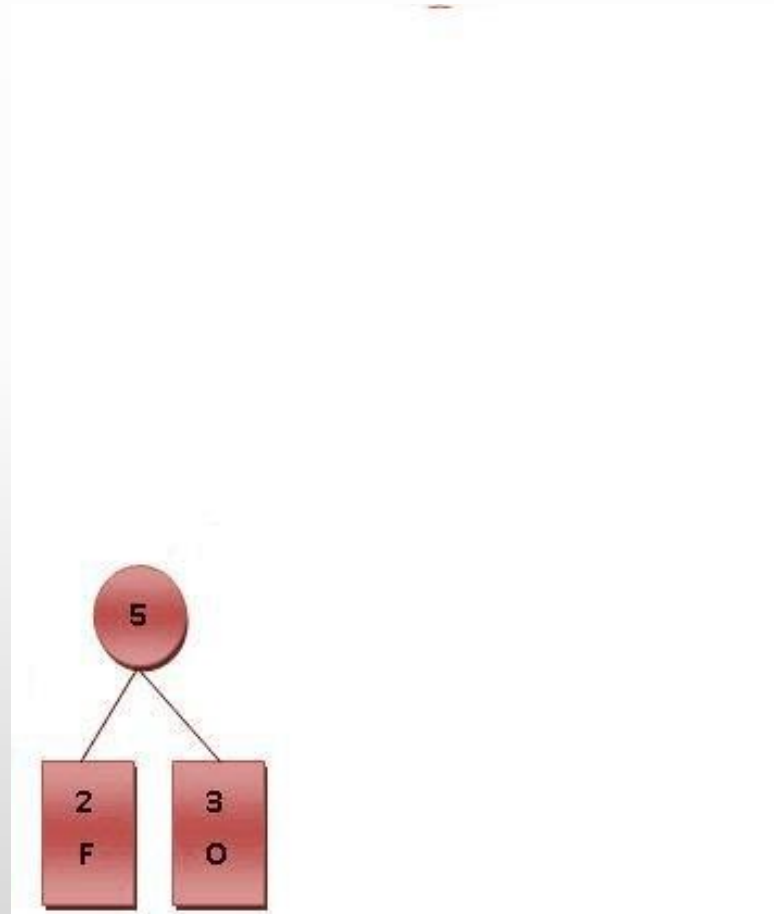


# Huffman Kodlama

- Adım 1: Her karakterin frekansı hesaplanır.
- Adım 2: Frekanslara göre bir Huffman ağacı oluşturulur.
- Adım 3: Ağaçtaki her karaktere karşılık gelen kodlar atanır.
- Adım 4: Veri, oluşturulan Huffman kodlarıyla kodlanır.

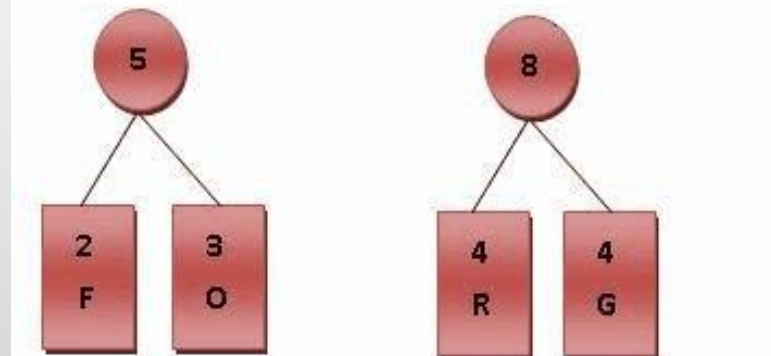


# Huffman Coding



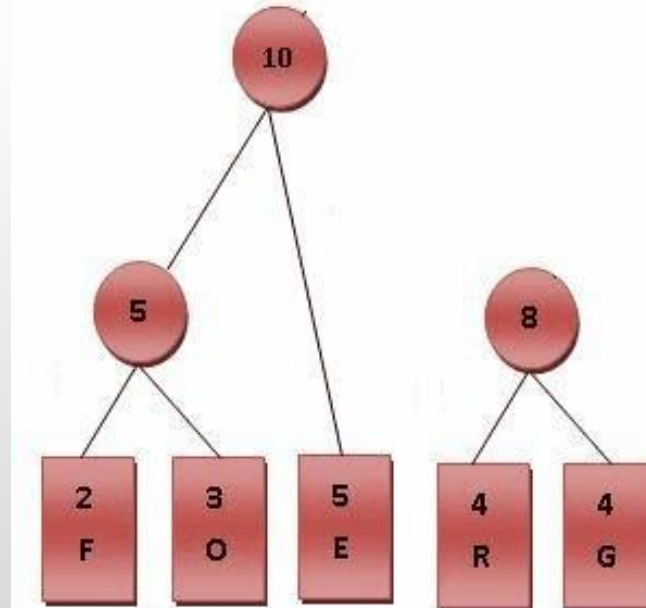


# Huffman Coding



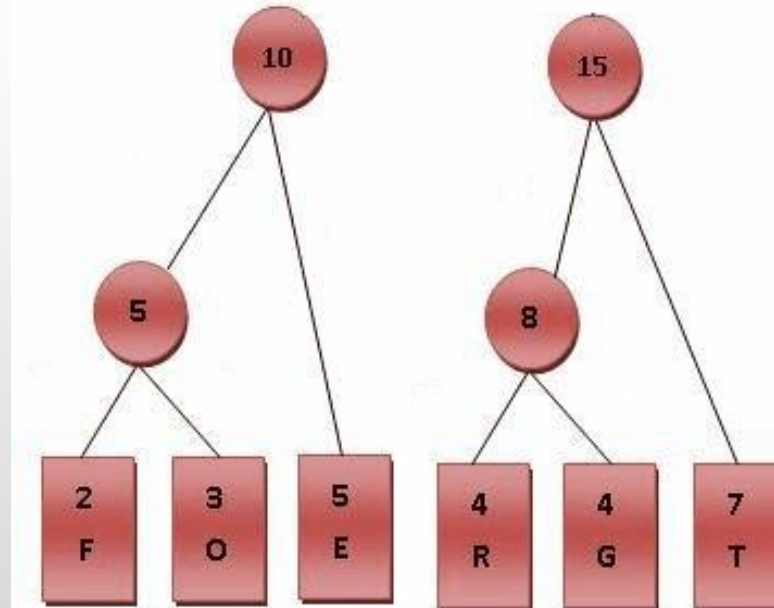


# Huffman Coding



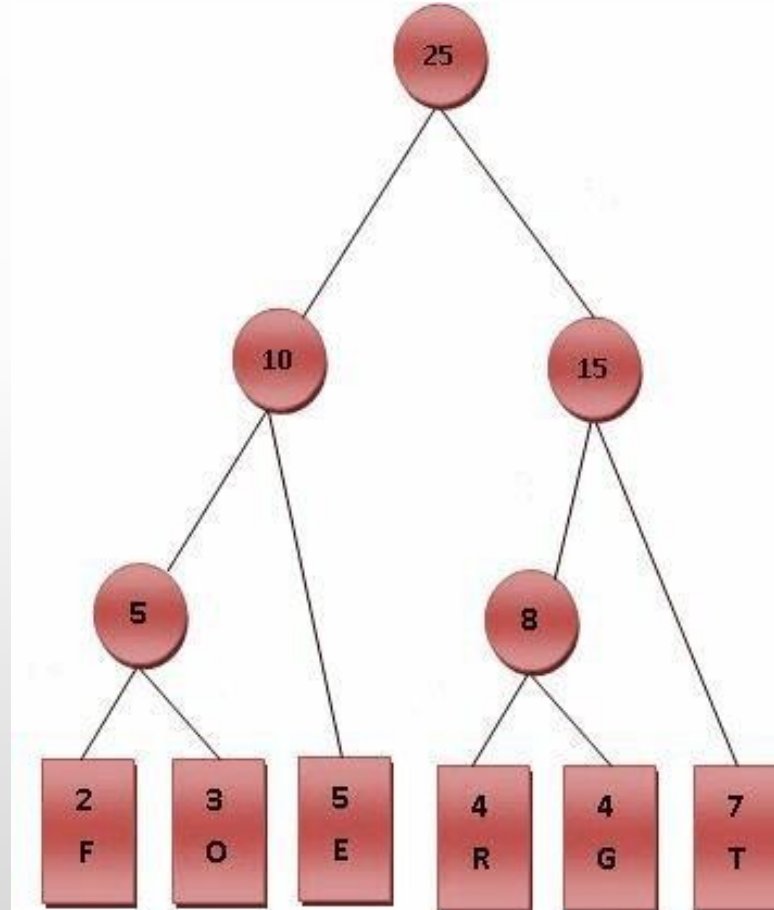


# Huffman Coding



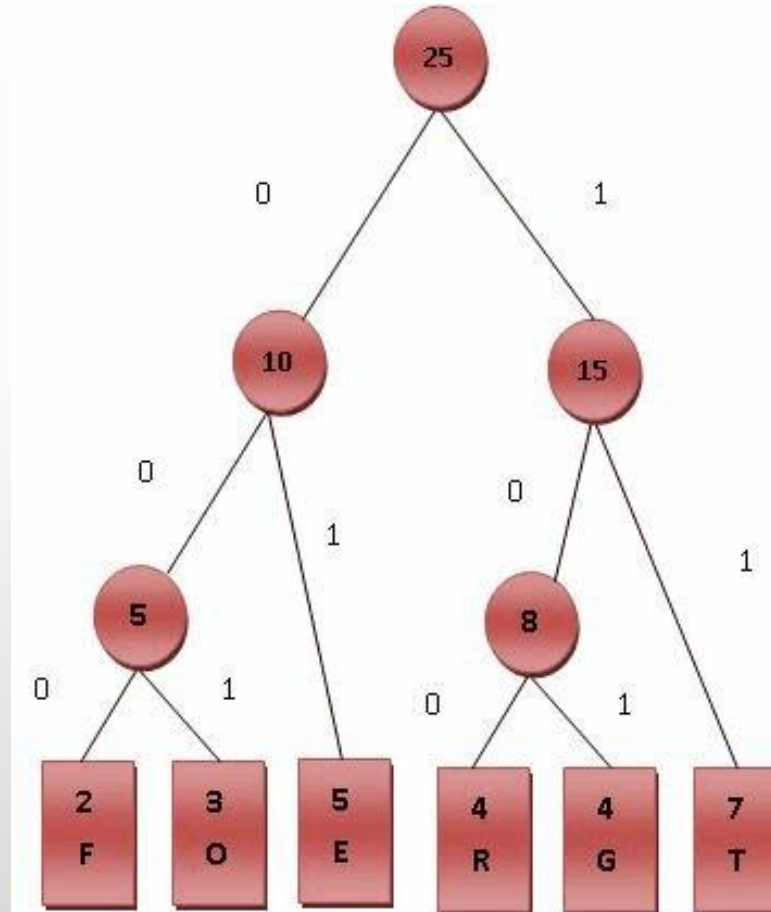


# Huffman Coding





# Huffman Coding









# Union-Find Veri Yapısı

- Amaç:
  - Küme birleştirme ve küme bulma işlemlerini hızlı bir şekilde yapmak.
- Operasyonlar:
  - Union(x, y):
    - x ve y kümesini birleştirir.
  - Find(x):
    - x elemanının bulunduğu kümenin kök elemanını bulur.



# Kruskal Algoritması

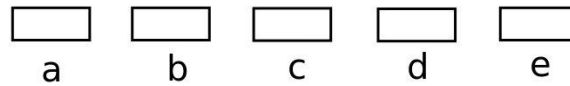
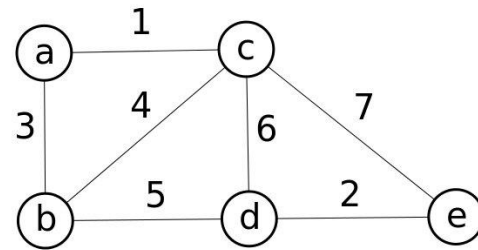
- Amaç:
  - Ağırlıklı bağlı bir çizgenin minimum ağaç oluşturma problemi.
- Greedy yaklaşım:
  - Her adımda en küçük ağırlığa sahip kenarı seçerek ilerler.
- İşleyiş:
  - Tüm kenarları ağırlıklarına göre sırala.
  - En küçük ağırlığa sahip kenarı seç.
  - Bu kenar, mevcut ağaçta döngü oluşturmaz ise ekle.
  - Tüm kenarlar eklenene kadar tekrarla.



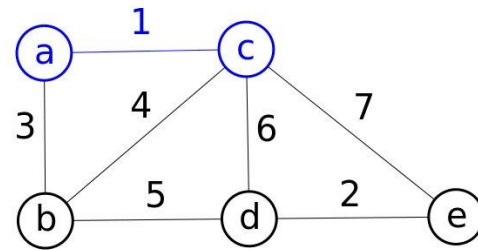
# Örnek

- D ğ mler: A, B, C, D, E, F
- Kenarlar: (A, B), (A, D), (B, C), (B, D), (C, E), (D, E), (D, F), (E, F)
- Ağırlıklar: 2, 4, 1, 3, 2, 1, 4, 3
- Kenarları Sırala: (C, E), (A, B), (D, E), (E, F), (A, D), (B, D), (D, F), (B, C)
- Ađaç Oluřtur: (C, E) -> (A, B) -> (D, E) -> (E, F)

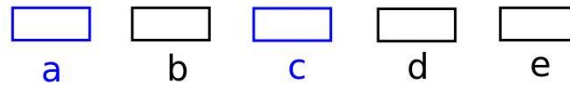
# Kruskal



# Kruskal

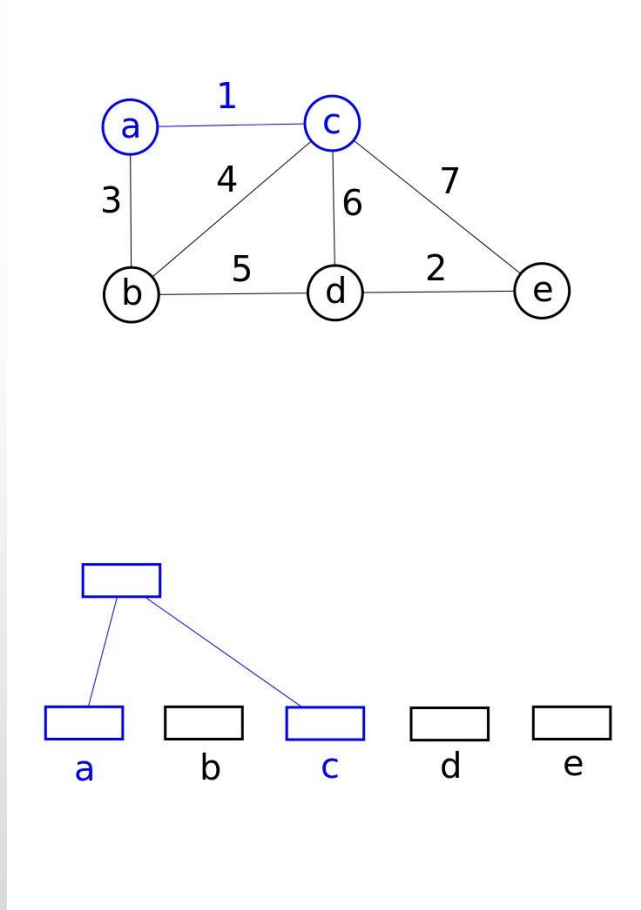


Disjoint!



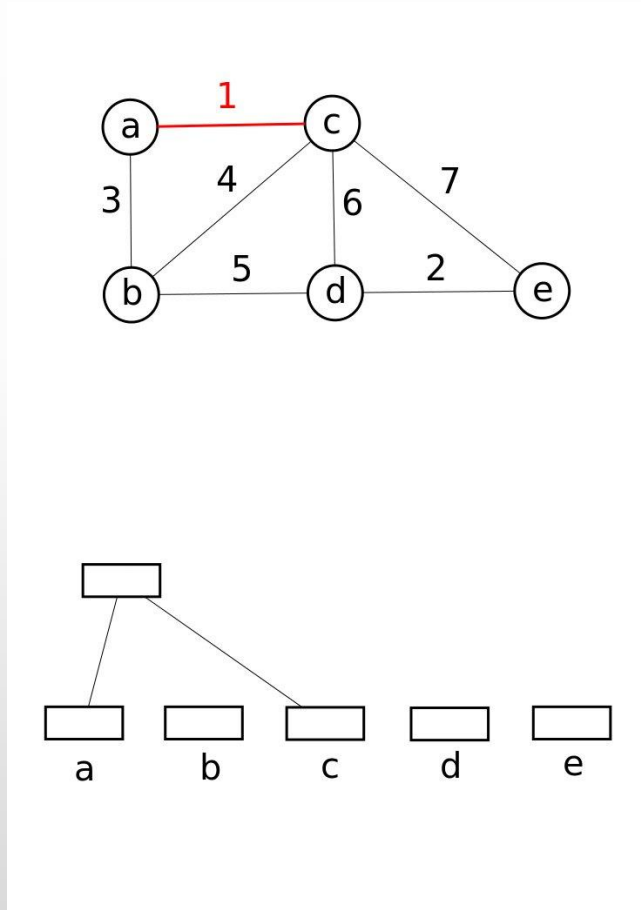


# Kruskal





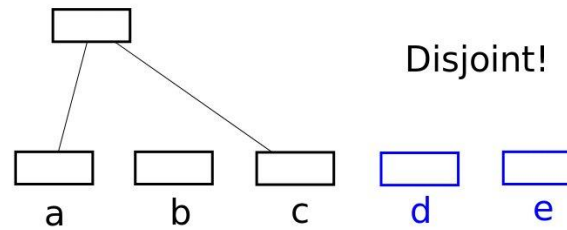
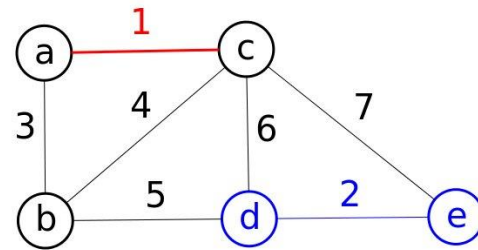
# Kruskal





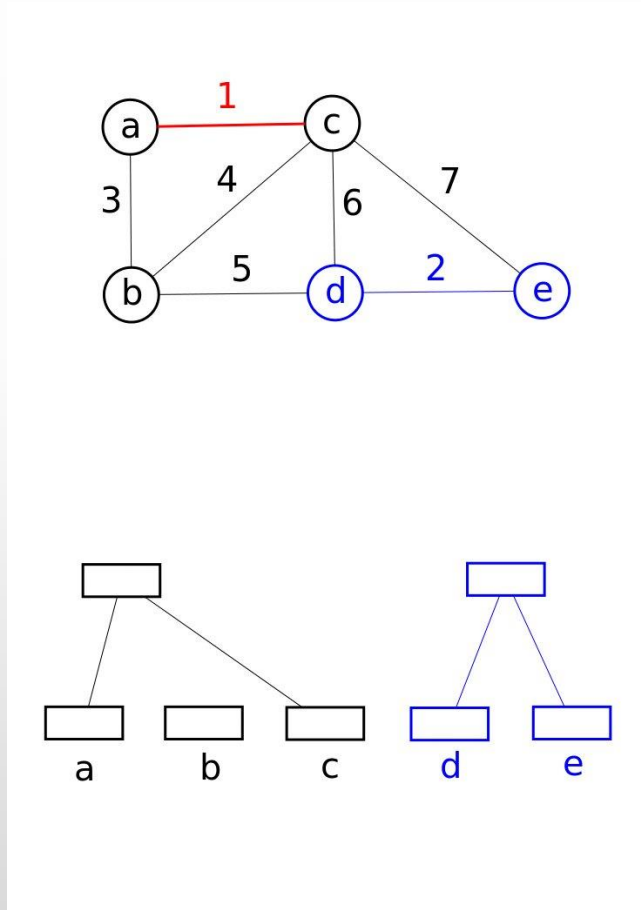


# Kruskal



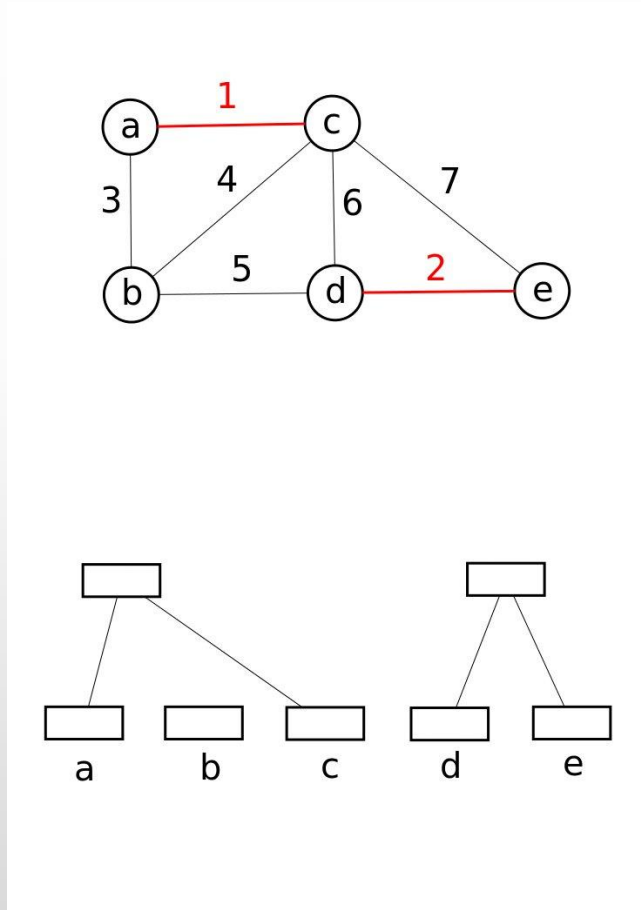


# Kruskal



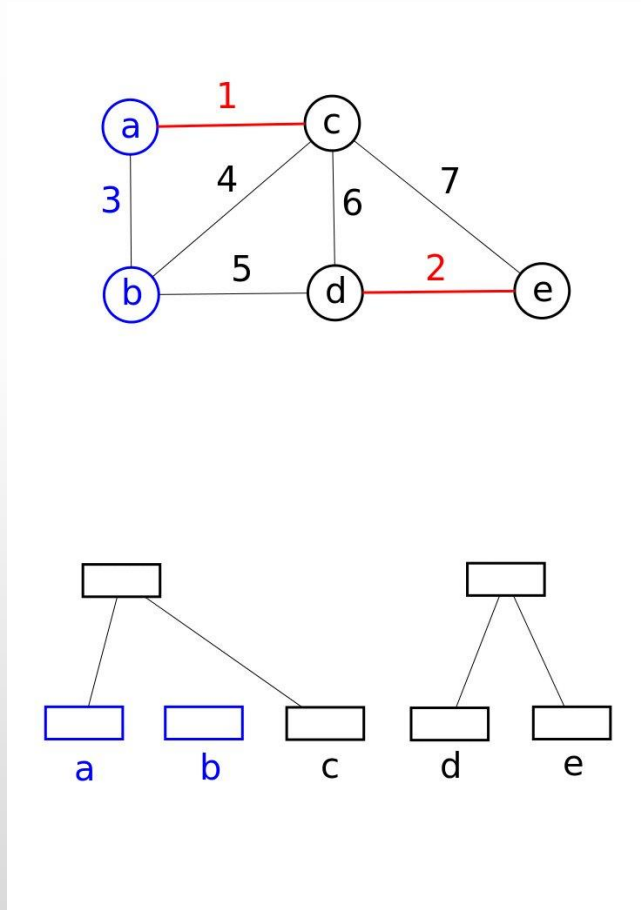


# Kruskal



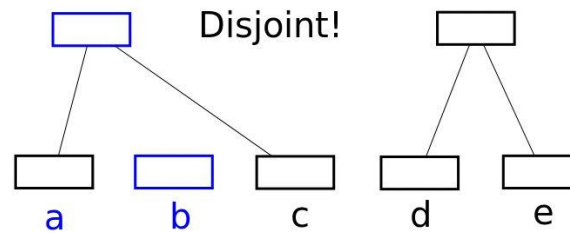
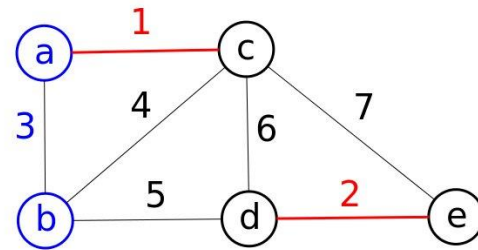


# Kruskal



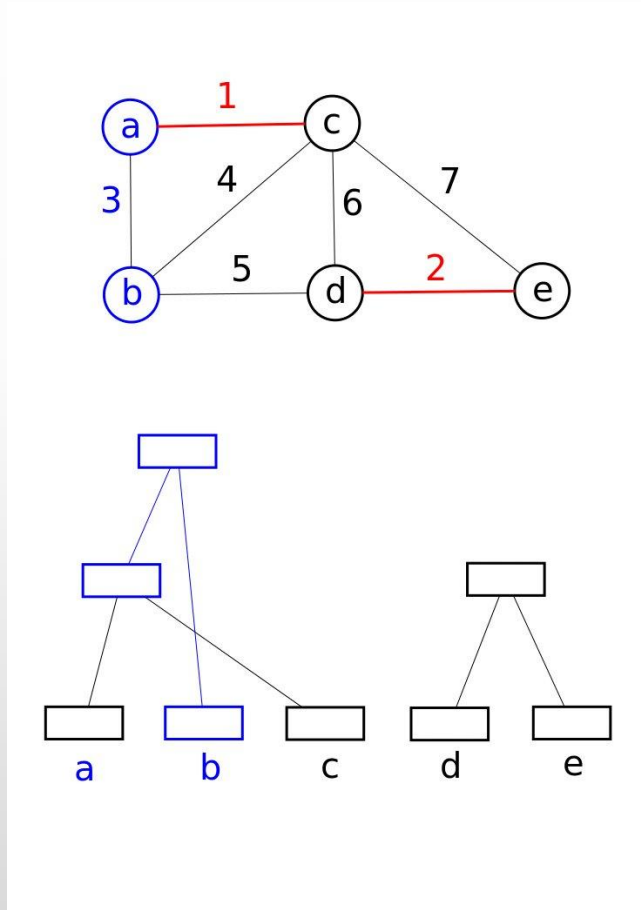


# Kruskal



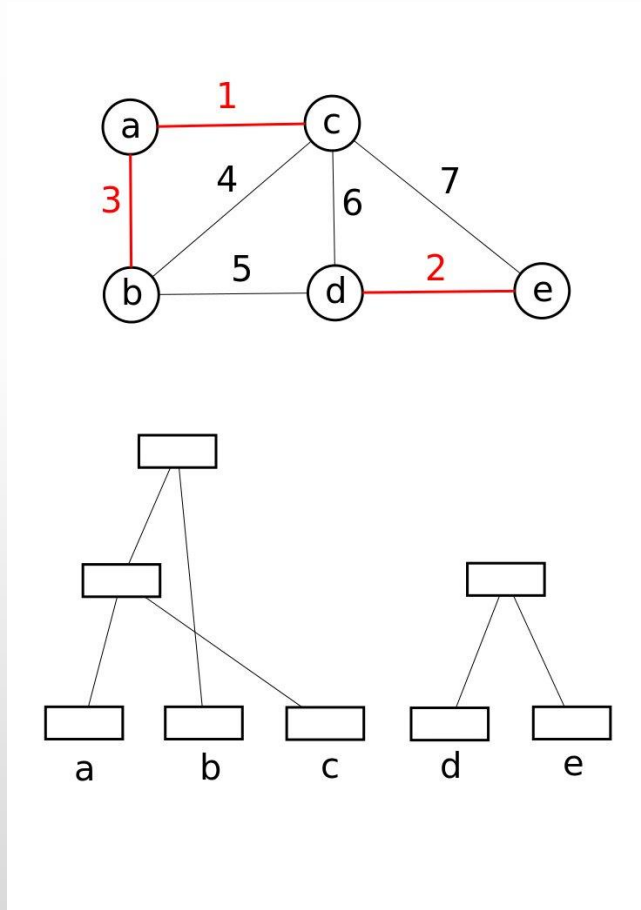


# Kruskal



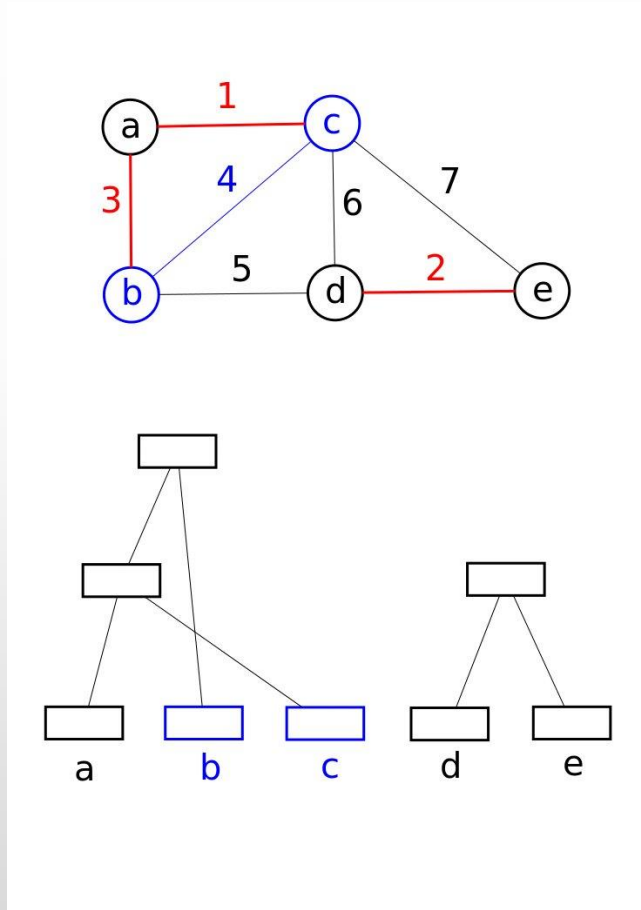


# Kruskal





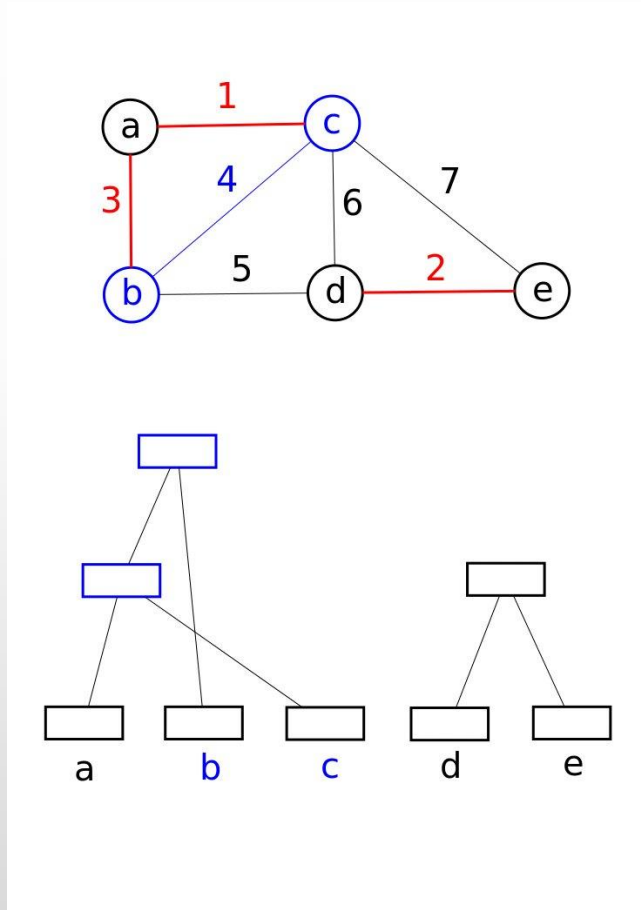
# Kruskal





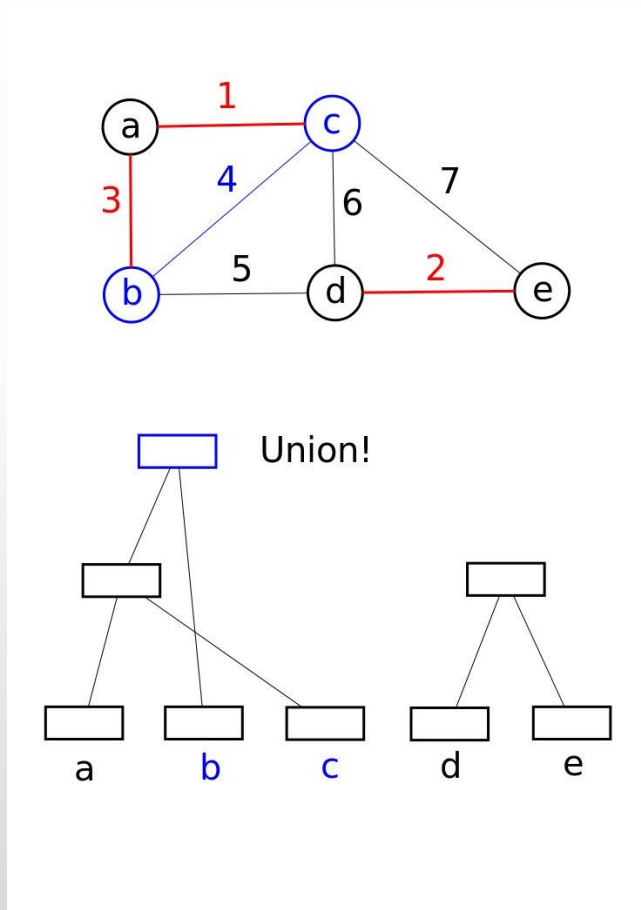


# Kruskal



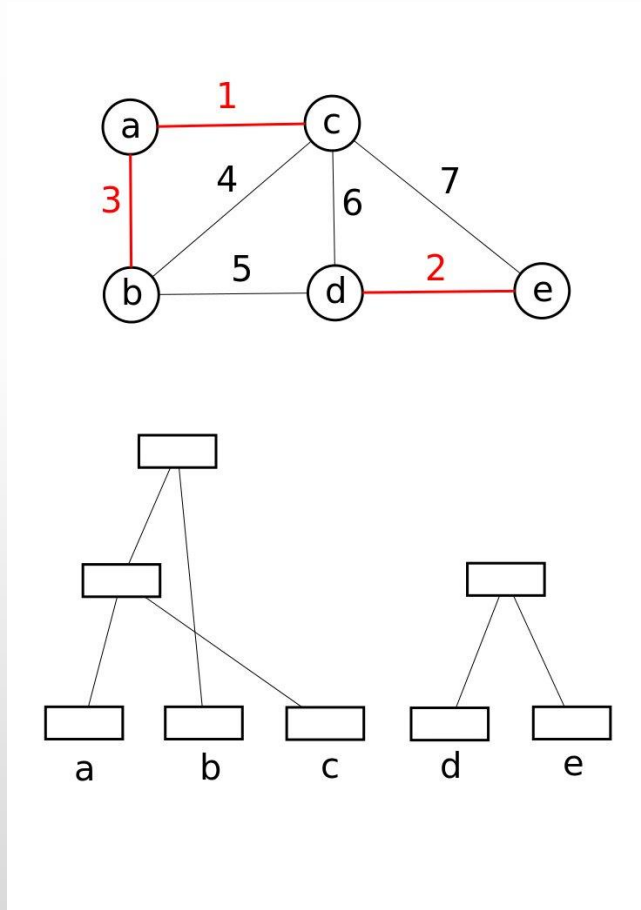


# Kruskal



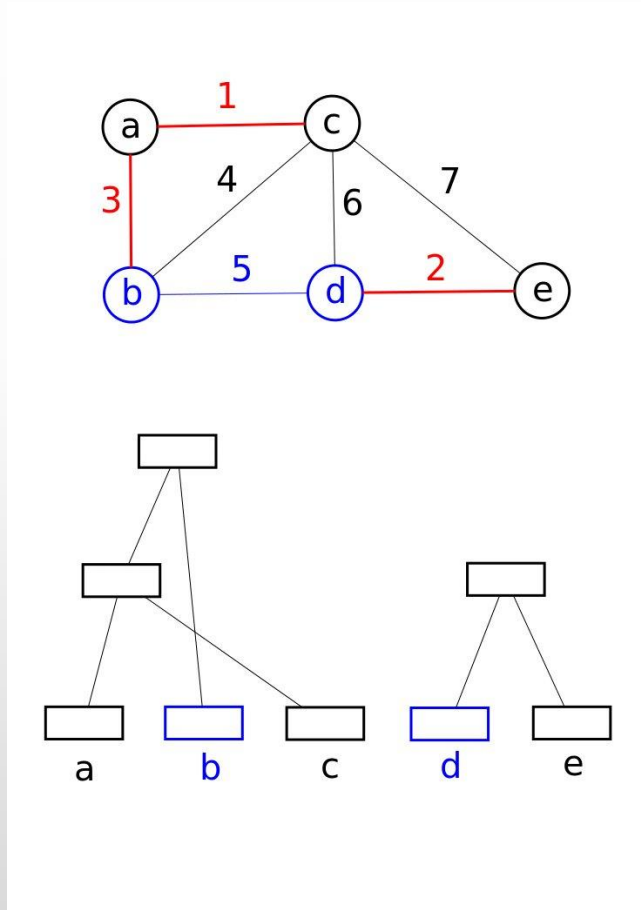


# Kruskal



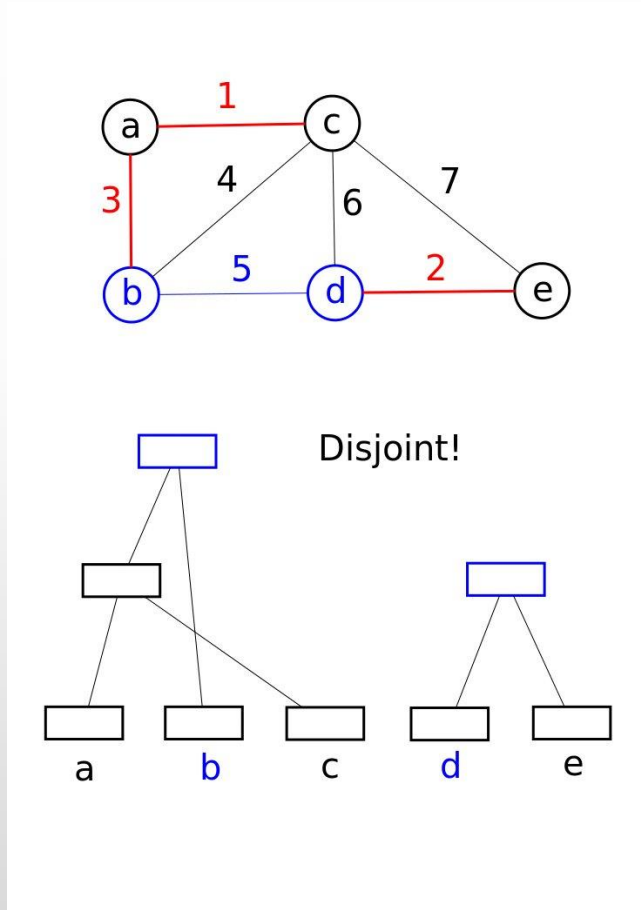


# Kruskal



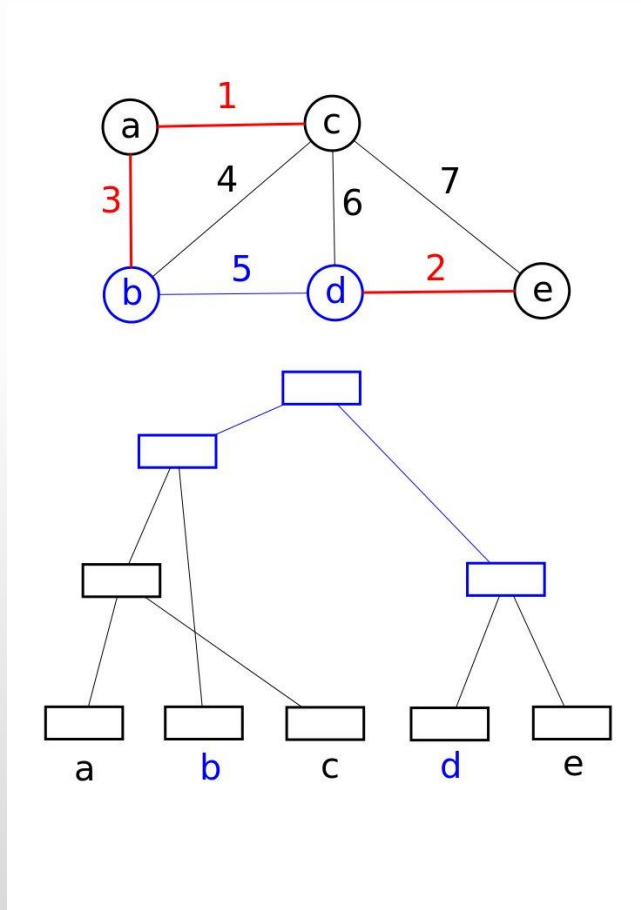


# Kruskal



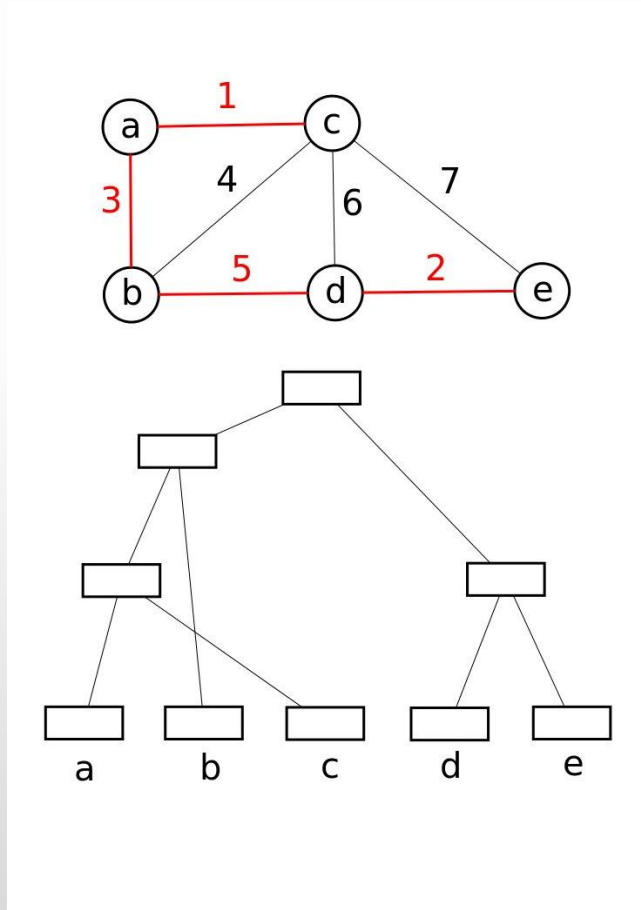


# Kruskal





# Kruskal









SON