

# **Çizge Algoritmaları**

## **İçerik**

- Çizge Temsili
- Genişlik Öncelikli Arama ( Breath First Search (BFS) )
- Derinlik Öncelikli Arama ( Depth First Search (DFS) )
- Topolojik Sıralama
- Güçlü Bağlı Bileşenler (Strongly Connected Components (SCC))
- Ayrık Küme İşlemleri
- Minimum Kapsama Ağacı (Minimum Spanning Tree (MST))
- En kısa yol problemi
- Tek kaynaklı en kısa yol bulma
- Tüm düğüm çiftleri arasında en kısa yol bulma

## Çizge Temsili

Bir çizge  $G = (V, E)$ , düğüm kümesi ( $V$ ) ve düğümler arasındaki bağlantıları gösteren kenar kümesinden ( $E$ ) oluşur.

- Çizgeler yönlü veya yönsüz olabilir.

Çizgelerin temsil edilmesi için iki temel yol vardır:

1. Komşuluk listesi
2. Komşuluk matrisi

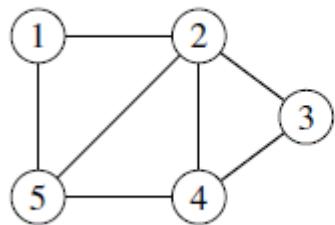
Bir algoritmanın çalışma zamanı genel olarak  $|V|$  ve  $|E|$  cinsinden ifade edilir. Yani düğüm ve kenar kümesinin eleman sayıları. Bunlar gösterilirken  $\mid$  işaretini de kullanılmayabilir. Örnek :  $O(V + E)$ .

Şimdi çizge temsil yöntemlerini inceleyelim.

### 1. Komşuluk listesi

Her bir düğümün komşusu listelenir. Düğüüm  $u$ 'nun listesi tüm düğümleri içerir öyle ki  $(u, v) \in E$ . (Bu temsil hem yönlü ve hem de yönsüz çizgeler için kullanılabilir.)

**Örnek:** Yönsüz bir çizge için



Adj	
1	2 /
2	1 /
3	2 /
4	2 /
5	4 /

Kenarların ağırlıkları varsa, bu ağırlıklar da listeye konulabilir.

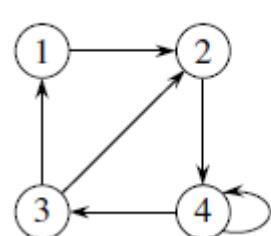
Ağırlık:  $w : E \rightarrow \mathbb{R}$

Yer:  $\theta(V+E)$

Zaman:  $u$ 'ya komşu olan tüm düğümleri listelemek :  $\theta(\text{derece}(u))$

Zaman:  $(u, v) \in E$  olup olmadığını bulmak :  $O(\text{derece}(u))$ .

**Örnek:** Yönlü bir çizge için



Adj	
1	2 /
2	4 /
3	1 /
4	4 /

Yönsüz bir çizge ile aynı asimptotik zaman ve yer karmaşıklığına sahiptir.

## 2. Komşuluk matrisi

$|V| \times |V|$  matrisi  $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{diğer durumlarda.} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Yukarıdaki örnek çizgeler için komşuluk matrisi bu şekilde görülür.

**Yer:**  $\theta(V^2)$

**Zaman:**  $u$ 'ya komşu olan tüm düğümleri listelemek :  $\theta(V)$

**Zaman:**  $(u, v) \in E$  olup olmadığını bulmak :  $\theta(1)$ .

Ağırlıklı çizge için (1-0) yerine ağırlıklarını da tutabilir.

## Genişlik Öncelikli Arama ( Breath First Search (BFS) )

**Girdi:** Çizge  $G = (V, E)$ , yönlü veya yönsüz, ve  $\text{başlangıç düğümü} s \in V$ .

**Cıktı:**  $d[v] = s$ 'den  $v$ 'ye uzaklık (en kısa kenar sayısı), tüm  $v \in V$  için.

### Çalışma mantığı:

Kaynak düğümden başla

Önce o düğümün çocuklarını ziyaret et

Sonra ziyaret edilen en son düğümün çocuklarını ziyaret vs.

Bu ziyaretler için FIFO (First IN First Out, İlk giren ilk çıkar) kuyruğu kullan.

Aşağıdaki algoritmada, her bir düğümün kaynak düşüğe olan uzaklığının yanında, ataları'da tutulmaktadır. (Algoritmada  $\text{pred}$  olarak belirtilmiştir.)

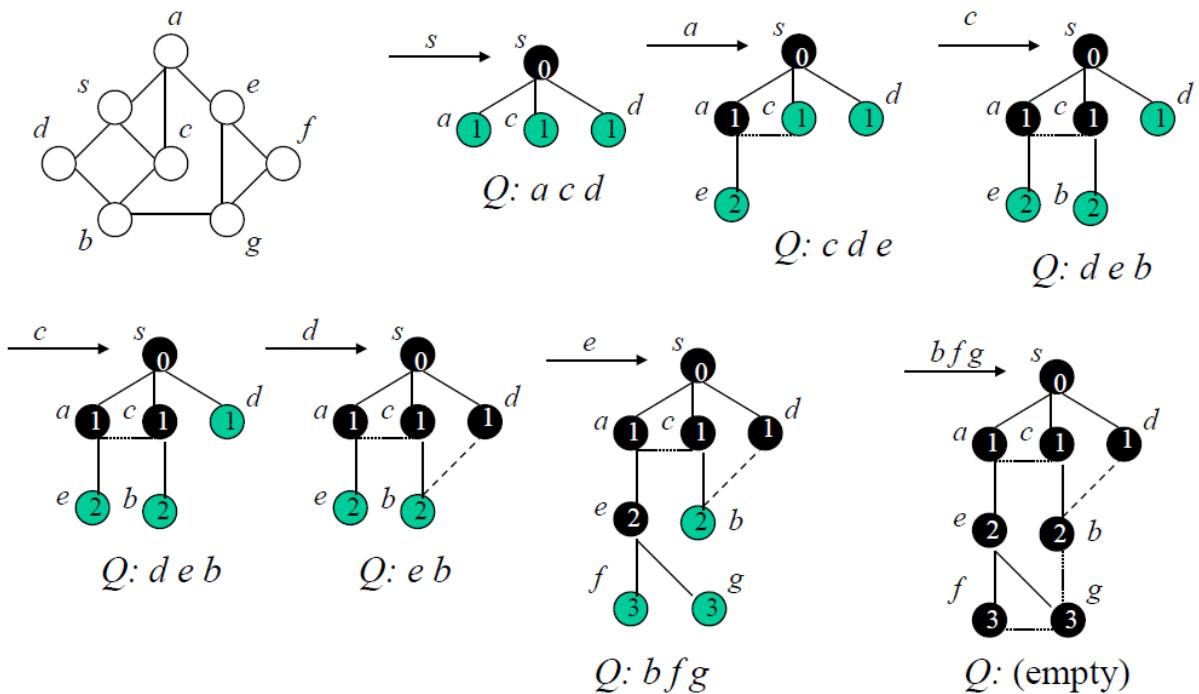
$\text{pred}[v] = u$  öyleki  $s$ 'den  $v$ 'ye giden yoldaki son kenar  $(u, v)$ 'dir.

- $u, v$ 'nin **atasıdır**.
- kenar kümesi  $\{(pred[v], v) : v \neq s\}$  ağaç yapısı oluşturur.

### BFS Algoritması:

```
BFS( $G, s$ ) {
    for each  $u$  in  $V$                                 // initialization
        { color[ $u$ ] = white; d[ $u$ ] = INFINITY; pred[ $u$ ] = NULL; }
    color[ $s$ ] = gray; d[ $s$ ] = 0; // initialize source  $s$ 
     $Q = \{s\}$ ;                                // put  $s$  in the queue
    while ( $Q$  is nonempty) {
         $u = \text{Dequeue}[Q]$ ;                      //  $u$  is the next vertex to visit
        for each  $v$  in  $\text{Adj}[u]$  {
            if (color[ $v$ ] == white) {                // if neighbor  $v$  undiscovered
                color[ $v$ ] = gray; d[ $v$ ] = d[ $u$ ]+1; pred[ $v$ ] =  $u$ ;
                Enqueue( $Q$ ,  $v$ );                  // ...put it in the queue
            }
        }
        color[ $u$ ] = black;                         // we are done with  $u$ 
    }
}
```

Örnek:



BFS Çalışma Zamanı:

$n=|V|$  ve  $m=|E|$  olsun. Başlangıç (initialization) için  $\Theta(n)$  zamana ihtiyacımız var. Tüm düğümler sadece bir defa ziyaret edildiği için while döngüsünün çalışma zamanı  $|V|$ 'dir. İçerideki for döngüsü derece( $u$ ) + 1 ile orantılı olarak çalıştırılacaktır. Buna göre çalışma zamanı şu şekildedir.

$$T(n, m) = n + \sum_{u \in V} (\deg(u) + 1) = n + n + 2m = \Theta(n + m)$$

## Derinlik Öncelikli Arama ( Depth First Search (DFS) )

**Girdi:** Çizge  $G = (V, E)$ , yönlü veya yönsüz. **başlangıç düğümü** verilmiyor !

**Çıktı:** Herbir düğüm için 2 tane zaman değeri :

- $d[v] = \text{keşif zamanı}$
- $f[v] = \text{sonlandırılma zamanı}$

Aynı zamanda BFS gibi  $\text{pred}[v]$  'de hesaplanır.

Her bir kenarı inceler.

Gerekli olduğu müddetçe farklı düğümlerden algoritma çalışır.

BFS'den farklı olarak yeni bir düğüm keşfettiğimizde, o düğümden keşifler başlar.

DFS algoritması çalışırken düğümlerin **renkleri** vardır:

- BEYAZ(WHITE) = keşfedilmemiş
- GRİ (GRAY) = keşfedilmiş, fakat henüz sonlandırılmamış (yani o düğümden başlayan keşif henüz bitmemiş)
- SİYAH (BLACK) = sonlandırılmış (o düğümden ulaşılabilen her şeyi bulduk)

Keşif ve sonlandırılma zamanları:

- 1 ile  $2|V|$  arasında değişen farklı sayılardır.
- Tüm  $v'$ ler için,  $d[v] < f[v]$ .

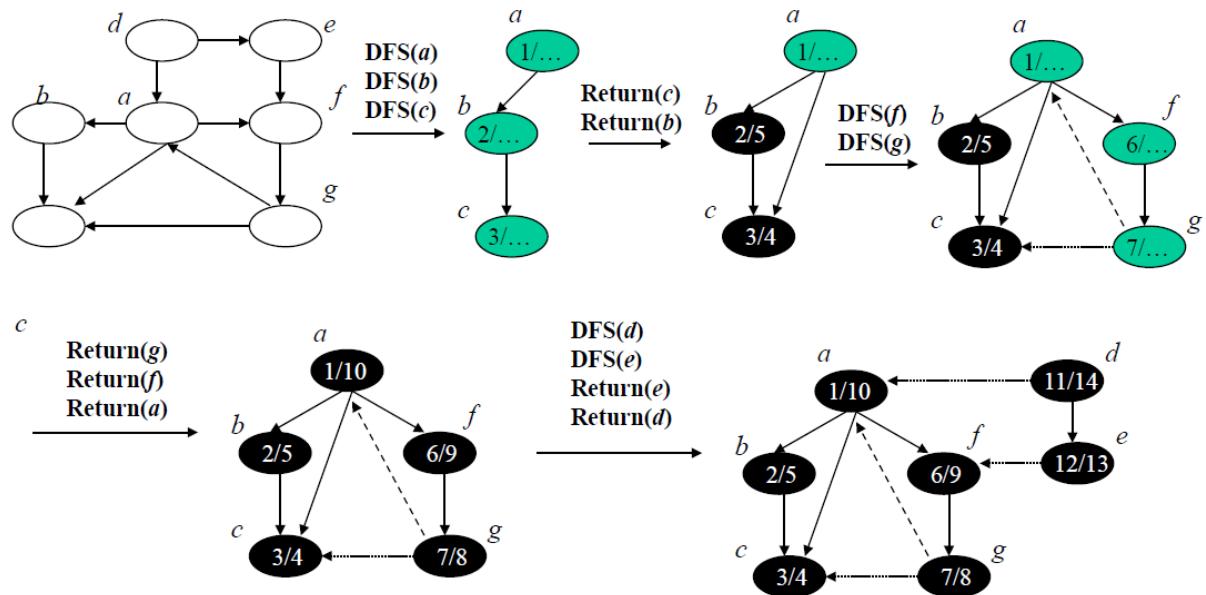
Başka bir deyişle,  $1 \leq d[v] < f[v] \leq 2|V|$ .

**DFS Algoritması:**

```
DFS(G) {
    for each  $u$  in  $V$  {
        color[ $u$ ] = white;
        pred[ $u$ ] = nil;
    }
    time = 0;
    for each  $u$  in  $V$ 
        if (color[ $u$ ] == white)
            DFSVisit( $u$ );
    }
}
```

```
DFSVisit( $u$ ) {
    color[ $u$ ] = gray;
     $d[u] = ++time$ ;
    for each  $v$  in Adj( $u$ ) do
        if (color[ $v$ ] == white) {
            pred[ $v$ ] =  $u$ ;
            DFSVisit( $v$ );
        }
    color[ $u$ ] = black;
     $f[u] = ++time$ ;
}
```

## Örnek:



## DFS Çalışma Zamanı:

BFS'de yaptığımız analize benzer şekilde çalışma zamanı  $\theta(V + E)$ 'dır.

## Parentez Teoremi

Tüm  $u, v$ 'ler için aşağıdakilerden tam olarak bir tanesi doğrudur:

- $d[u] < f[u] < d[v] < f[v]$  or  $d[v] < f[v] < d[u] < f[u]$  ve  $u$  ve  $v$  birbirlerinin çocukları veya torunları (descendant) değildir. (Birbirlerinin neslinden gelmezler)
- $d[u] < d[v] < f[v] < f[u]$  ve  $u, v$ 'nin atasıdır.
- $d[v] < d[u] < f[u] < f[v]$  ve  $v, u$ 'nun atasıdır.

Yani,  $d[u] < d[v] < f[u] < f[v]$  ilişkisi olamaz.

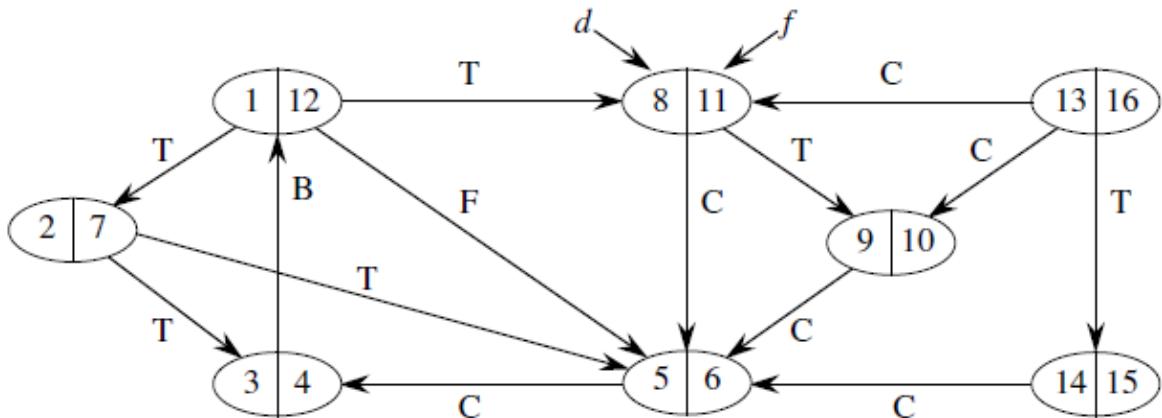
Parantezler gibi:

- OLABİLİR: () [] ([ ]) [ () ]
- OLAMAZ: ( [ ) [ ( ) ]

## Kenarların Sınıflandırılması

- Ağaç Kenar (Tree edge):** derinlik öncelikli orman içinde bulunan kenarlar.  $(u, v)$  kenarını keşfederken bulunan kenarlardır.
- Ters Kenar (Back edge):**  $(u, v)$ ,  $v, u$ 'nın atasıdır.
- İleri Kenar (Forward edge):**  $(u, v)$ ,  $v, u$ 'nın neslindendir fakat ağaç kenar değildir.
- Çapraz Kenar (Cross edge):** Diğer tüm kenarlar. Aynı derinlik öncelikli ağaçlar içinde veya farklı derinlik öncelikli ağaçlar içinde bulunabilir.

Örnek:



**Teorem:** Yönsüz bir çizgenin DFS’inde sadece ağaç ve ters kenarlar bulunur. İleri veya çapraz kenarlar bulunmaz.

DFS’ın her bir düğüm için hesapladığı zaman değerleri çizge ile ilgili bir çok bilgi verir. Örneğin, çizgede döngü olup olmadığını bulabiliriz.

Verilen bir yönlü çizge  $G = (V, E)$ ’nin DFS ağaç kümelerini ve bu kümedeki  $(u, v)$  kenarını göz önüne alın.

- Eğer bu kenar ağaç, ileri veya çapraz kenar ise  $f[u] > f[v]$ ’dir.
- Eğer bu kenar ters kenar ise  $f[u] \leq f[v]$ ’dir.

**Teorem:**  $G$ ’de döndü vardır ancak ve ancak  $G$ ’nin DFS ağaç kümelerinde ters kenar var ise.

# Topolojik Sıralama

## Yönlü ve Döngüsüz Çizge (Directed Acyclic Graph (DAG) )

Yönlü ve döngüsüz çizgeler, önceliğin önemli olduğu bir çok uygulamada karşımıza çıkmaktadır. Genel olarak, bir DAG içinde düğümler işleri, kenarlar ise bu işler arasındaki sıralamaları veya öncelikleri belirler.  $(u,v)$  kenarı  $v$  işinin başlamadan önce mutlaka  $u$ 'nun bitirilmesi gerektiğini göstermektedir.

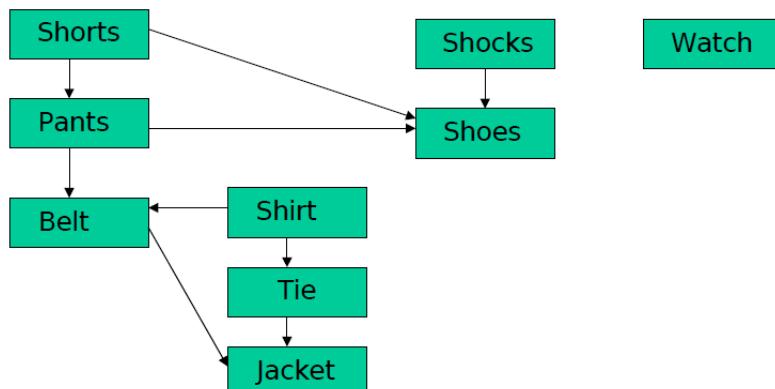
Kısmi sıralaması olan işlemleri veya yapıları modellemede kullanılabilir.

- $a > b$  ve  $b > c \rightarrow a > c$ .
- Fakat aralarında  $a > b$  veya  $b > c$  ilişkilerinin olmadığı a ve b 'ler olabilir.

Kısmi sıralamadan (tüm  $a \neq b$ 'ler içina  $>$   $b$  ya da  $b > a$ ) tam sıralama elde edilebilir.

Topolojik sıralama, tam sıralamayı elde etmek için kullanılmaktadır.

**Örnek:** Bir kişinin üstünü giymesindeki öncelik sıralaması şu şekilde olabilir:



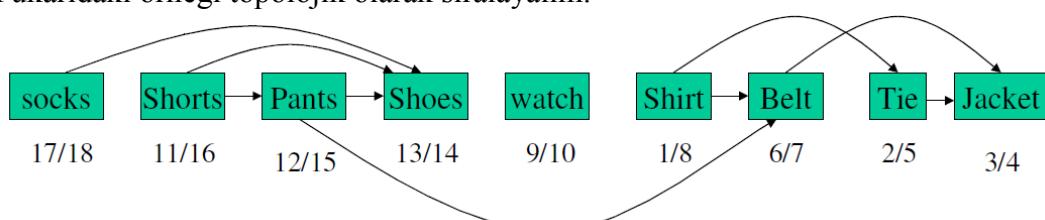
Bir DAG'de topolojik sıralama yapıldığında, o DAG'in düğümlerinin doğrusal sıralaması yapılır. Eğer  $(u,v) \in E$  ise  $u$ , sıralamada  $v$ 'den önce bir yererde görülmeliidir. (sayıların sıralanması gibi değil!)

TOPOLOJİK-Sıralama( $G=(V,E)$ )

1.  $DFS$ 'i çalıştır.
2.  $DFS$  algoritmasında bir düğüm sonlandırıldığında bağlı listenin başına yerleştir.
3. Bağlı listeyi *return* et.

**Çalışma Zamanı:**  $\Theta(V + E)$ 'dir.

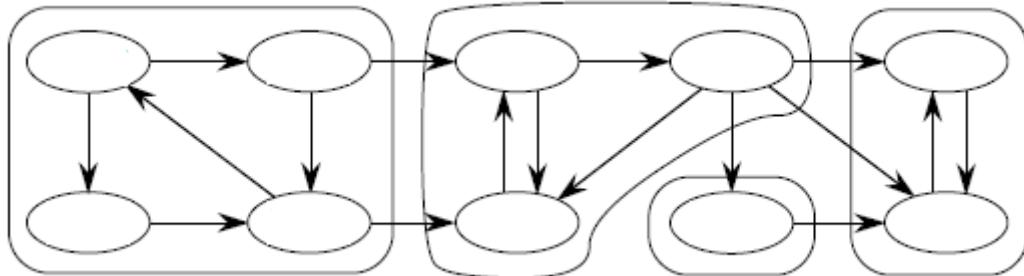
Yukarıdaki örneği topolojik olarak sıralayalım.



## Güçlü Bağlı Bileşenler (Strongly Connected Components (SCC))

Verilen bir yönlü çizgede  $G = (V, E)$ ,  $G$ 'nin SCC'si en büyük düğüm kümeleridir  $C \subseteq V$  öyleki tüm  $u, v \in C$  için, hem  $u$ 'dan  $v$ 'ye ve hem de  $v$ 'den  $u$ 'ya bir yol vardır.

Örnek:



Algoritma,  $G^T = G$ 'nin **tersini** kullanır.

- $G^T = (V, E^T)$ ,  $E^T = \{(u, v) : (v, u) \in E\}$ .
- $G^T$ ,  $G$ 'nin tüm kenarları ters çevrilmiş halidir.

$G^T$ , komşuluk listesinin kullanılması ile  $\Theta(V + E)$  zamanda hesaplanabilir.

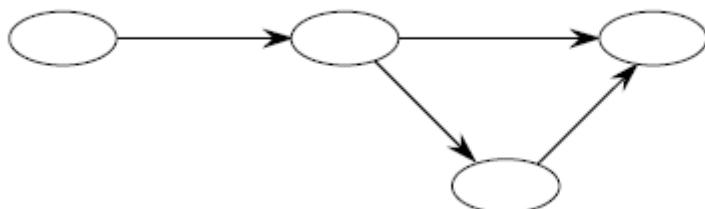
$G$  ve  $G^T$  aynı SCC'ye sahiptir. ( $u$  ve  $v$  birbirlerine  $G$  içinde ulaşabilirler ancak ve ancak  $G^T$  içinde birbirlerine ulaşabilirlerse.)

### Bileşen Çizge

$G_{SCC} = (V_{SCC}, E_{SCC})$ .

- $V_{SCC}$ ,  $G$ 'deki her bir SCC için bir tane düğüm içerir.
- $E_{SCC}$  iki düğüm arasında bir kenara sahiptir eğer  $G$ 'de o düğümlere karşılık gelen SCC'ler arasında bir kenar varsa.

Yukarıdaki örnek için  $G_{SCC}$  şu şekildedir:



**Lema:**  $G_{SCC}$  bir DAG'dır. Daha biçimsel olarak,  $G$  çizgesindeki iki farklı SCC,  $C$  ve  $C'$  olsun.  $u, v \in C$  ve  $u', v' \in C'$ 'olduğunu kabul edelim. Eğer  $G$ 'de  $u$  ile  $u'$  arasında bir yol var ise,  $u$  ile  $u'$  arasında bir yol olamaz.

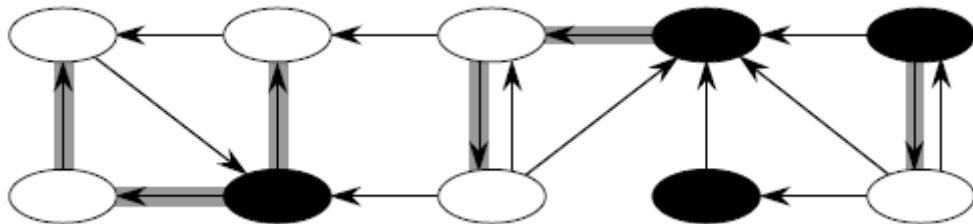
$SCC(G)$

1.  $DFS(G)$  algoritmasını çağır tüm  $u$  düğümü için sonlandırılma zamanını ( $f[u]$ ) hesapla
2.  $G^T$ 'yi bul

3. DFS(GT) algoritmasını çağır fakat ana döngüde, düğümleri azalan  $f[u]$  sıralamasına göre al
4. İlkinci DFS algoritmasının bulduğu genişlik-öncelikli ağaç kümelerinin her bir ağaçını faktı SCC olarak yaz.

**Örnek:**

1. DFS
2.  $G^T$
3. DFS (ağaçın kökleri siyaha boyanmıştır)



**Çalışma Zamanı:**  $\theta(V + E)$ 'dir.

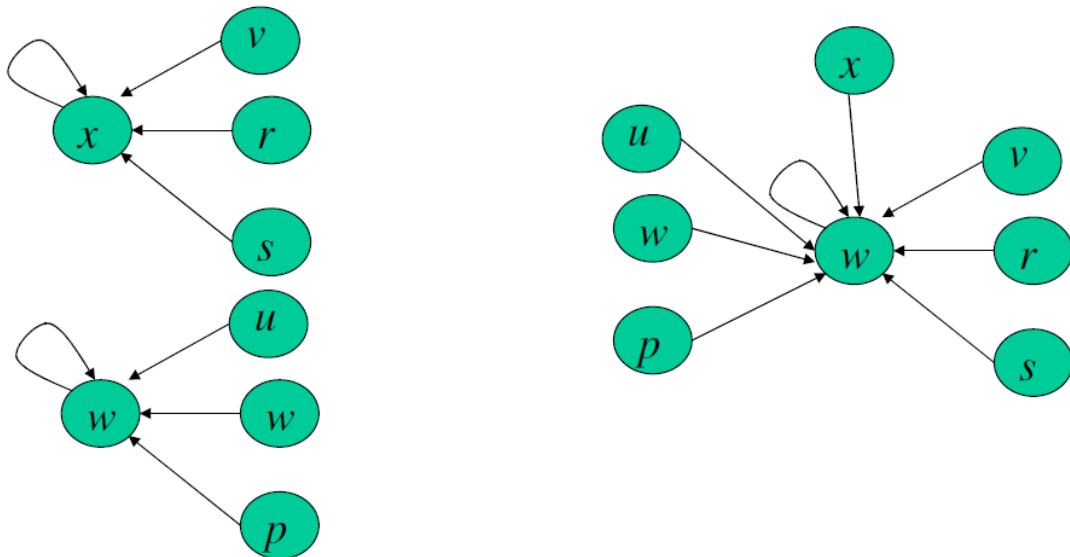
## Ayrık Küme İşlemleri

- $\text{MakeSet}(v)$ :  $v$  elemanını içeren bir küme oluşturur,  $\{v\}$ .
- $\text{FindSet}(u)$ :  $u$ 'nin dahil olduğu kümeyi return eder.
- $\text{Union}(u,v)$ :  $u$ 'nun ve  $v$ 'nin dahil olduğu iki kümeyi birleştirir.

**Örnek:**

Bir kümeyi göstermek için bağlı liste kullanalım.

- $\text{Make-Set}(w)$ :
- $\text{Find-Set}(u)$ : ( $w$  return eder)
- $\text{Union}(u,v)$ :



Ayrık küme işlemlerinin bir çizgenin bağlı bileşenlerini (connected components) ve iki düğümün  $u, v$  aynı bileşende olup olmadığını bulmak için kullanılabilir:

**Bağlı\_Bileşenler( $G$ )**

for each vertex  $v \in V[G]$

    MAKESET( $v$ )

for each edge  $(u,v) \in E[G]$

    if FINDSET( $u$ )  $\neq$  FINDSET( $v$ )

        UNION( $u,v$ )

**Aynı\_Bileşen( $u,v$ )**

if FINDSET( $u$ ) = FINDSET( $v$ )

    return TRUE

else     return FALSE

**Çalışma Zamanı:**

Make-Set ve Find-Set  $O(1)$  zamanda çalışır. Peki Union ?

Bunun için farklı bir soru soralıım.  $N=\{1,\dots,n\}$  kümesi  $n$  tane tam sayıdan oluşsun.

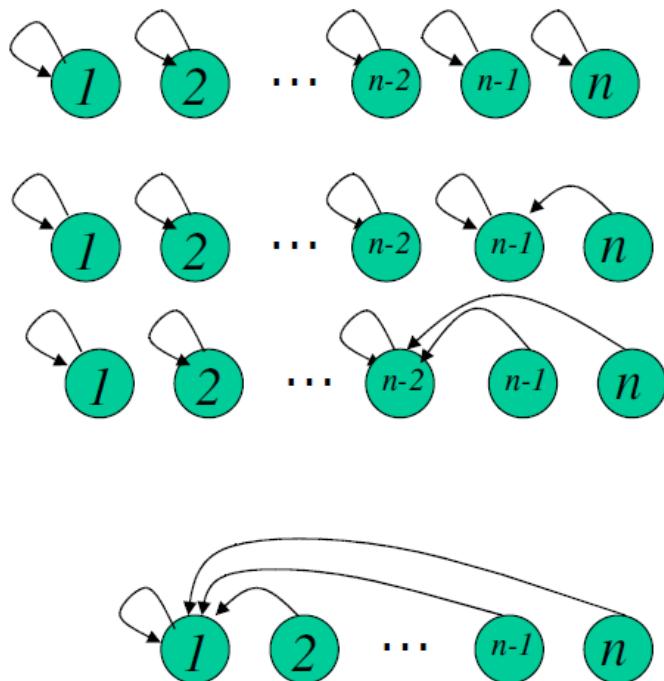
$P = \{(u, v) \mid u \text{ ve } v \in N\}$  kümesinde  $N \times N$ 'den gelen ikilileri içersin. Buna göre,  
for  $u=1$  to  $n$  MakeSet( $u$ );

```
for every pair  $(u, v) \in P$ 
    if FindSet( $u$ ) ≠ FindSet( $v$ )
        Union( $u, v$ )
```

yukarıdaki kodda bir elemanın pointer'i en fazla ne kadar değişir ?

Cevap:  $O(n^2)$ . Neden ?

Aşağıdaki şekil bunu açıklıyor.

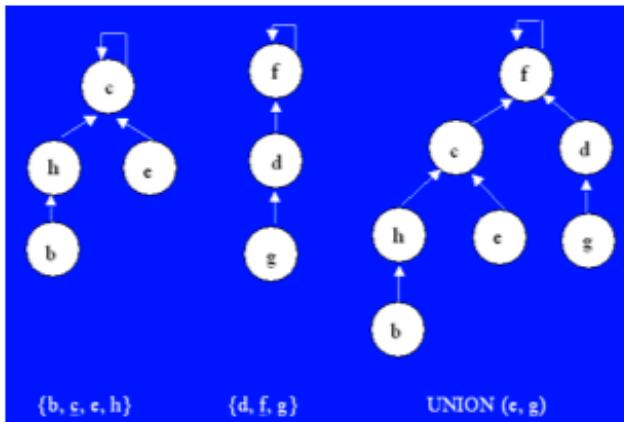


- Her kümede, o kümenin eleman sayısını o kümenin kökünde (veya temsilcisinde) tutalım. Bu sayı Rank( $u$ ) ile belirtilsin.
- Buna göre iki küme birleştirildiğinde her zaman rank'i küçük olan kümenin pointer'ları değiştirilsin.
- Bu durumda, bir pointer değiştiğinde kümenin eleman sayısı en az 2 kat artar.
- Tüm bu işlemler sonunda, en büyük kümenin eleman sayısı  $n$ 'dir.
- Bir pointer'ın en fazla değişme sayısı da  $\log n$ 'dır.
- $n$ -eleman önüne alındığında toplamda  $O(n \log n)$  değişiklik vardır.

### Ayrık Küme Ağaçları

- Her bir ağaç bir kümeyi temsil eder.
- Her bir düğüm bir tane üye içerir.
- Her bir üye sadece kendi atasını gösterir.
- Her bir kök, temsilcisi içerir ve her kök kendisinin atasıdır.

### Ayrık küme ağaçları için Örnek:



### Ayrık Küme İşlemlerinin Gerçekleştirilmesi:

- MAKE-SET(x): Elemanı sadece x olan yeni bir ağaç oluşturur.
- FIND-SET(x): x'den başlayarak ata pointer'larını kök düğümüne kadar takip et, kök düğümünü return et.
- UNION(x,y): Bir ağaçın kök pointer'ını diğerinin kök düğümünü gösterecek şekilde değiştir.

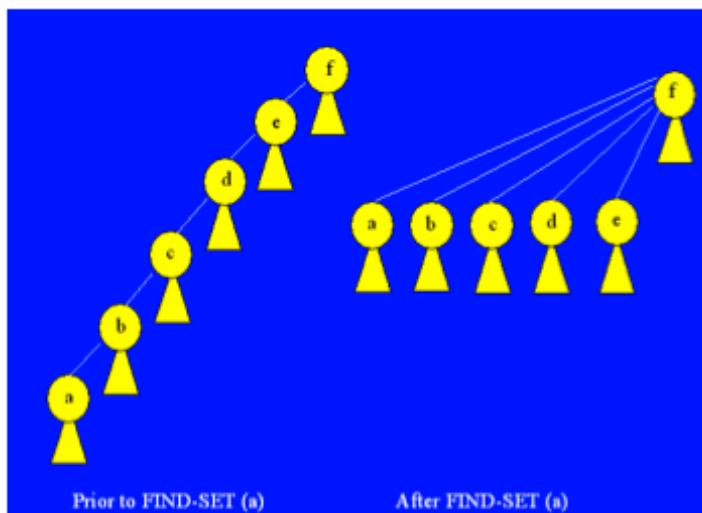
Şuandaki hali ile çok iyi değil, doğrusal zincir şeklinde düğümler elde edilebilir. Fakat, aşağıdaki iki tane sezgisel yöntem (heuristic) ile birlikte kullanıldığında ayrık küme işlemlerinin çalışma zamanı iyileşirilebilir.

1. **Rank ile union:** Birleşme işlemi çalıştırıldığında, küçük kümenin kök düğümünün pointer'ını büyük kümenin kök'ünü gösterecek şekilde değiştir.

Eleman sayısını kullanmaz. Rank'ı kullanır, rank burada ağaçın yüksekliğini gösterir.

2. **Yol sıkıştırması:** Find-Set işlemi çalıştırıldığında, find-path üzerindeki her bir düğümün direk olarak kök düğümünü göstermesini sağla.

### Yol sıkıştırma için Örnek:



### Rank ile Union:

Her bir düğüm x'in  $\text{rank}(x)$ , ( $x$ 'in rank'i), ve  $p(x)$  ( $x$ 'in atası) bulunur.

- **Make-Set( $x$ )** { $p[x] \leftarrow x$ ;  $rank[x] \leftarrow 0$ }
- **Union( $x, y$ )** {**Link(Find-Set( $x$ ); Find-Set( $y$ ))**}
- **Link( $x, y$ )**
  - if  $rank[x] > rank[y]$  then  $p[y] \leftarrow x$
  - else  $p[x] \leftarrow y$
  - if  $rank[x] = rank[y]$  then  $rank[y] \leftarrow rank[y] + 1$

#### Yol Sıkıştırması:

Find-Set işlemi özyinelemeli olarak tanımlanır. Böylece, find-path üzerindeki düğümlerin pointer'ları değiştirilir.

- **Find-Set( $x$ )**
  - if  $x \neq p[x]$
  - then  $p[x] \leftarrow \text{Find-Set}(p[x])$
  - return  $p[x]$

**Çalışma Zamanı:** Rank ile birleşme ve yol sıkıştırma yöntemleri beraber kullanıldığında çalışma zamanı  $O(m \alpha(n))$  olmaktadır.  $n$ 'ye karşı  $\alpha(n)$  şu şekilde değişmektedir.

$n$	$\alpha(n)$
0-2	0
3	1
4-7	2
8-2047	3
2048-A <sub>4</sub> (1)	4

A<sub>4</sub>(1) nedir? Bu değer  $10^{80}$ 'den büyüktür. (Yani evrendeki tüm atomların sayısından).

## Minimum Kapsama Ağacı (Minimum Spanning Tree (MST))

$G=(V,E)$ ,  $n$  tane düğümü,  $m$  tane kenarı ve kenarlarının ağırlıkları ( $w$ ) bulunan bir çizge olsun.

- İçerisinde döngü barındırmayan ve tüm düğümleri içeren bağlı bir alt çizgeye ( $T$ ), o çizgenin kapsama ağacı denir.
- $T$ 'nin ağırlığı,  $T$ 'nin içindeki tüm kenarların ağırlıklarının toplamıdır:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

- Kapsama ağaçları içinde ağırlığı minimum olan ağaç, o çizgenin minimum kapsama ağacıdır.

### Genel MST Algoritması

- Kenarları ağırlıklarına göre sırala
- Sıralı listedeki her bir kenar, eğer daha önceden ağaca eklenmiş olan kenarlar ile bir döngü oluşturmuyorsa ağaca ekle. Diğer durumlarda ekleme.

Algoritma  $n-1$  tanekenareklendiğindesonlandırılabilir.

Step 1 'in çalışmazamanı  $O(m \log m) = O(n \log n)$ .

Step 2 ise  $O(n \log n)$  zamanalır.

Ancak iyileştirilmiş yapıyı kümeleri ile bu adımda doğrusal zamanda yapılabilir.

### Kruskal'ın MST Algoritması

- Bu algoritma, direk olarak genel MST algoritması üzerine kurulmuştur.
- Her bir iterasyonda, ağaca eklenmesi güvenli olan min. ağırlıklı kenar bulunur ve  $A$  kümesine eklenir. Algoritmanın sonunda bu kümeye MST olur.
- Algoritmanın çalışması esnasında  $A$ , bir ağaç kümesini (orman) oluşturur.

```
1.  $A \leftarrow \emptyset$ 
2. for each  $v \in V_G$  do
3.   Make - Set( $v$ )
4. Sort Edges in  $E_G$ 
5. for each  $(u,v) \in E_G$ 
   (artan sırada kenarları al)
6.   if Find - Set( $u$ ) ≠ Find - Set( $v$ )
7.      $A \leftarrow A \cup \{(u,v)\}$ 
8.     Union( $u,v$ )
9. Return  $A$ 
```

### Çalışma Zamanı:

Step 1:  $O(1)$

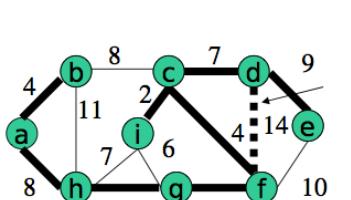
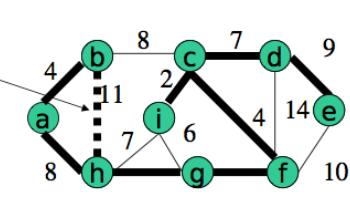
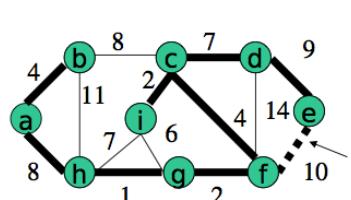
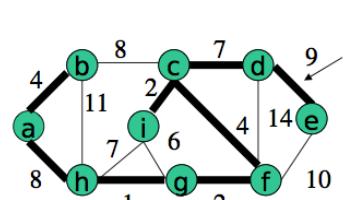
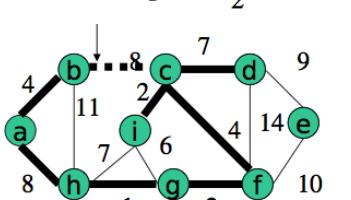
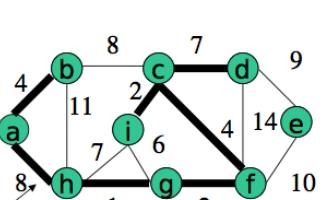
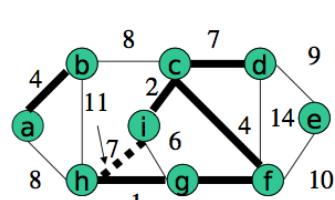
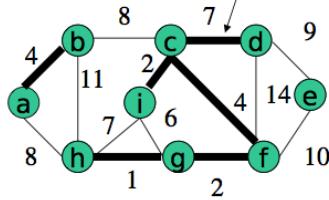
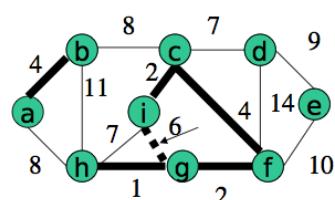
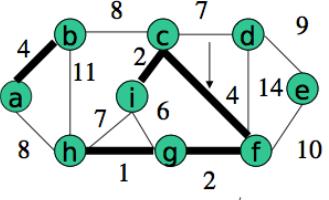
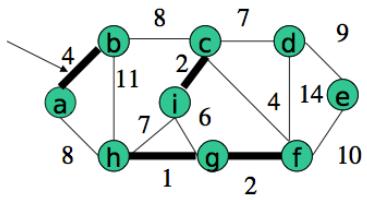
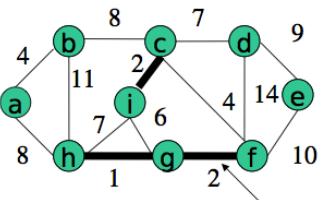
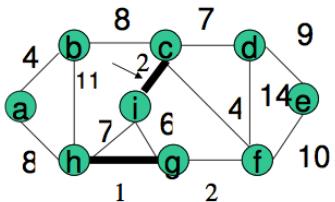
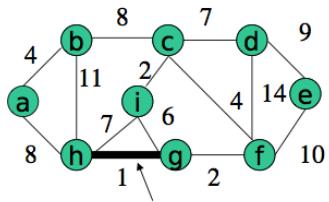
Steps 2,3:  $O(n)$

Step 4:  $O(m \log m)$

Steps 5-8:  $O(m \log n)$

Eğer Rank ile birleşme ve yol sıkıştırması yöntemleri birlikte kullanılırsa, Steps 5-8 doğrusal zamanda yapılabilir. Genel olarak bu algoritmanın çalışma zamanı  $O(m \log m)$  'dir.

Örnek:



## Prim'in MST Algoritması

- Bu algoritma, direk olarak genel MST algoritması üzerine kurulmuştur.
- Her adımda,  $A$  tek bir bağlı bileşendir.
- Algoritma, rasgele bir  $r$  düğümünden başlar (ağaçın kök düğümü).
- Her adımda,  $A$ 'ya en küçük ağırlıklı kenar ile bağlı yeni bir düğüm eklenir.
- Büyüme  $r$ 'den başlar ve tüm düğümler kapsanana kadar devam eder. Her  $u$  düğümünün atası  $p[u]$  vardır.
- Ayrıca, her  $u$  düğümünün değeri ( $\text{key}[u]$ ) vardır. Bu değer,  $u$ 'nun  $A$ 'ya eklenmesinin maliyetini belirler.

MST – Prim( $G, W, r$ )

1.  $Q \leftarrow V_G$
2. For each  $u \in Q$ 
  3. do  $\text{key}[u] = \infty$
  4.  $\text{key}[r] \leftarrow 0$
  5.  $p(r) = \text{NIL}$
  6. While  $Q \neq \emptyset$  do
    7.  $u \leftarrow \text{Extract - Min}(Q)$
    8. For each  $v \in \text{Adj}[u]$  do
      9. If  $v \in Q$  and  $w(u, v) < \text{key}[v]$ 
        10. then  $p(v) \leftarrow u$
        11.  $\text{key}[v] \leftarrow w(u, v)$

## Çalışma Zamanı:

Algoritmada, öncelik kuyruğunu gerçekleştirmek için binary heap kullanılabilir.

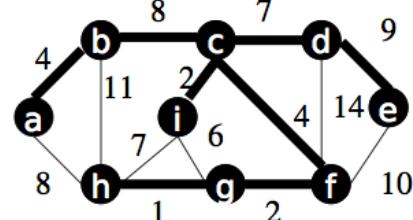
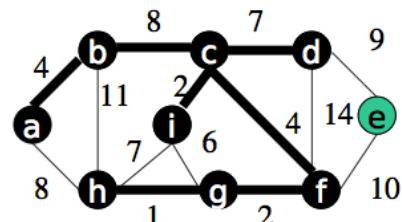
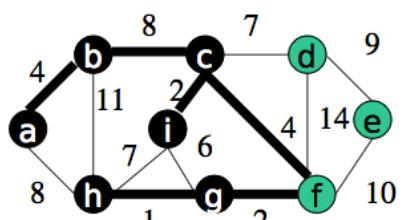
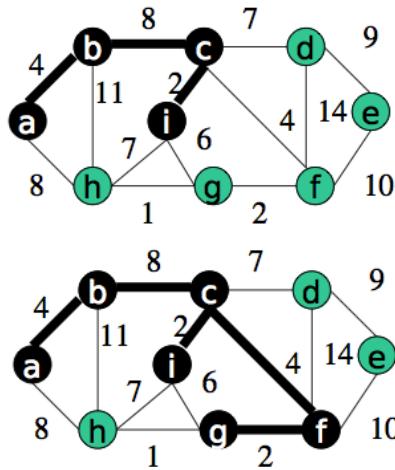
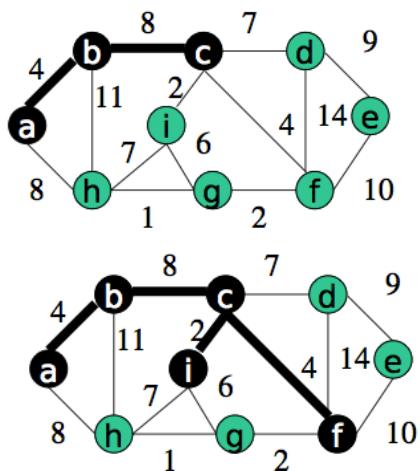
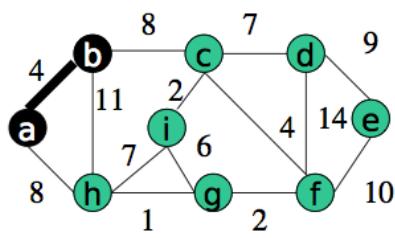
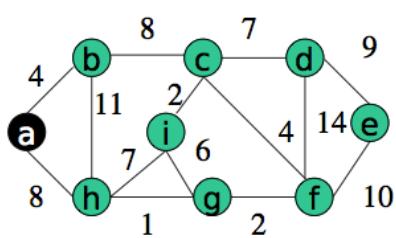
Steps 1-5:  $O(n)$

Step 7:  $O(\log n)$

Step 11:  $O(\log n)$  (decrease key işlemi ile)

$Q$ 'daki tüm elemanların  $\text{Adj}[]$  listesinde (yani tüm elemanların komşularının toplamı)  $m = |E|$  olduğu için algoritmanın toplam çalışma zamanı:  $O(n \log n + m \log n) = O(m \log n)$ .

Örnek:



## En Kısa Yol Problemi

Bir çizgedeki en kısa yolu bulma problemi bir çok uygulamada karşımıza çıkmaktadır. Taşıma problemi, robot hareketi planlama, ve iletişim problemleri bunlardan bazılarıdır. Bu problem, bir haritadaki iki nokta arasındaki en kısa yolu bulunması şeklinde tanımlanmaktadır.

### Girdi:

Yönlü bir çizge  $G = (V, E)$

Ağırlık fonksiyonu  $w : E \rightarrow \mathbb{R}$

Bir  $p$  yolunun ağırlığı  $p = \langle v_0, v_1, \dots, v_k \rangle$

$$= \sum_{i=1}^k w(v_{i-1}, v_i)$$

(o yolu üzerindeki tüm kenarların ağırlıklarının toplamı)

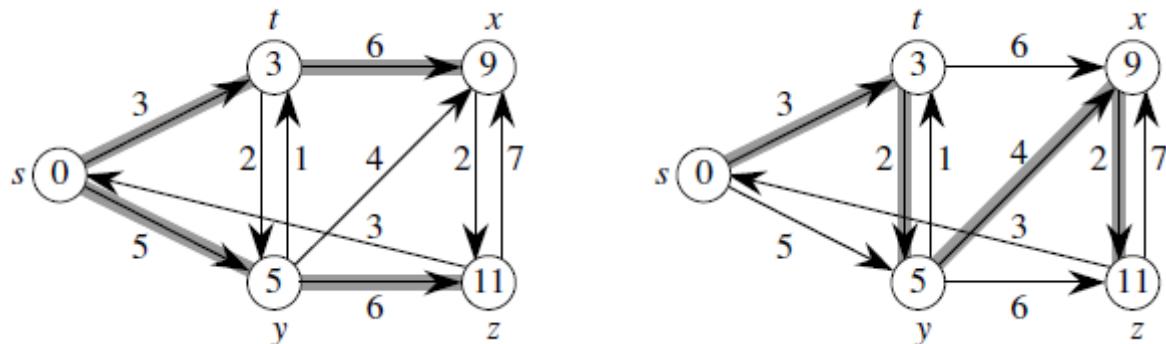
$u$ 'dan  $v$ 'ye en kısa yol ağırlığı:

$$\delta(u, v) = \begin{cases} \min \{w(p) : u \xrightarrow{p} v\} & \text{eğer } u \text{'dan } v \text{'ye bir yol var ise,} \\ \infty & \text{diğer durumlarda.} \end{cases}$$

$u$ 'dan  $v$ 'ye en kısa yol, her hangi bir  $p$  yoludur öyleki  $w(p) = \delta(u, v)$ .

### Örnek:

$s$ 'den diğer düğümlere en kısa yol.



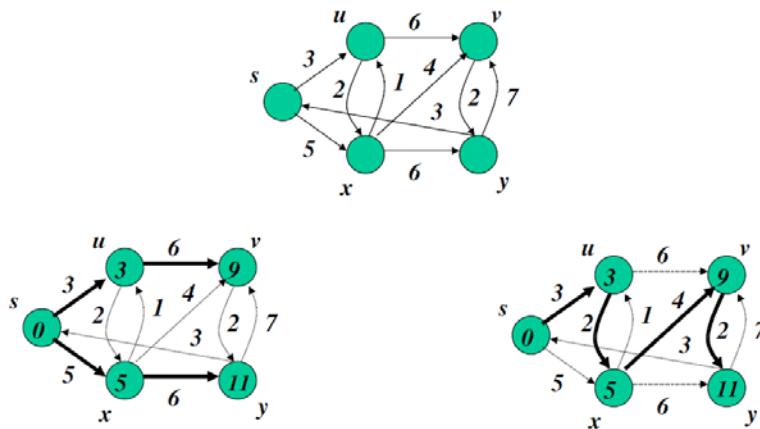
Bir düğümden diğer düğümlere en kısa yollar hesaplanırken, bu yollar bir ağaç yapısı içinde görülür.

### En kısa yol problem çeşitleri:

- Tek kaynaklı en kısa yol
- Tek hedefli en kısa yol
- Tek çift için en kısa yol
- Tüm çiftler için en kısa yol

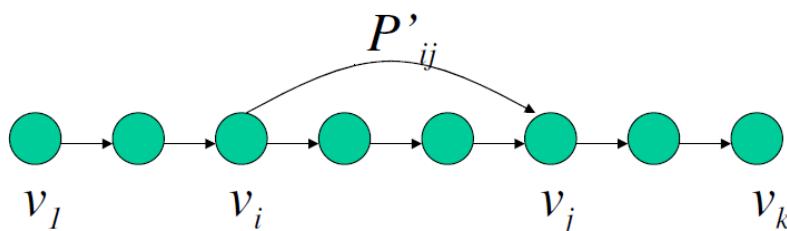
### Teklik Özelliği (Uniqueness):

İki düğüm arasındaki en kısa yol, birden fazla olabilir.



### Alt optimallik özelliği (sub-optimality)

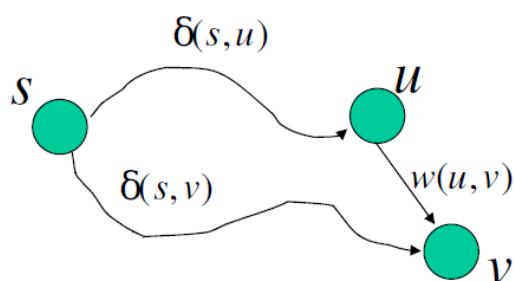
Verilen bir ağırlıklı çizge  $G=(V,E)$  ve kenarlar üzerinde tanımlı bir ağırlık fonksiyonu  $w$  için  $v_1$  ile  $v_k$  arasındaki en kısa yol  $p = \langle v_1, \dots, v_k \rangle$  olsun. Herhangi bir  $i$  ve  $j$  için  $p_{ij} = \langle v_i, \dots, v_j \rangle$ , bu yol  $(p)$  üzerindeki  $v_i$  ile  $v_j$  arasındaki yol olsun. Bu durumda  $p_{ij}$  yolu  $v_i$  ile  $v_j$  arasındaki en kısa yoldur.



Bu özellik kolay bir şekilde olmayana ergi (çelişki ile ispat) metodu ile görülebilir.

### Üçgen eşitsizlik özelliği:

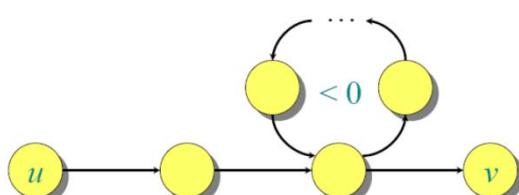
$G=(V,E)$  ağırlıklı bir çizge ve  $w$  kenarlar üzerinde tanımlı bir ağırlık fonksiyonu olsun. Buna göre, çizgedeki tüm kenarlar  $(u,v)$ , için  $\delta(s,v) \leq \delta(s,u) + w(u,v)$ 'dır.



### Negatif ağırlıklı döngüler:

Eğer çizgede negatif ağırlıklı döngüler varsa, bu çizgede bazı en kısa yollar mevcut değildir.

**Örnek:**



### Dijkstra'nın en kısa yol Algoritması:

- Bu algoritma, sadece kenar ağırlıkları negatif olmayan çizgeler için çalışır.
- Eğer çizge ağırlıksız ise bu algoritmanın sonucu BFS'e benzer.
- Prim'in algoritması gibi öncelik kuyruğu kullanır.
- Gevşeme(Relaxation): Her bir düğüm, derece sayısı kadar gevşeme işlemine tabidir.
- Gevşeme işlemi sonucundaki değişiklikler DecreaseKey algoritması tarafından ele alınır.

SSSP(G)

for each  $v \in V$  do

$$d[v] = \infty$$

$$d[s] = 0$$

$$S = \emptyset$$

$$Q = V$$

while  $Q \neq \emptyset$  do

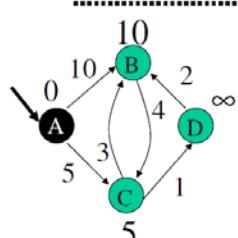
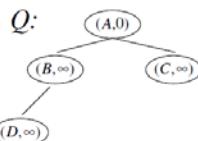
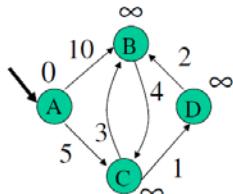
$$u = \text{Extract - Min}(Q)$$

$$S = S \cup \{u\}$$

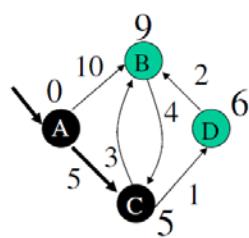
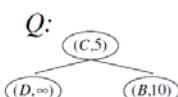
for each  $v \in \text{Adj}[u]$  do

if  $d[v] > d[u] + w(u, v)$  then

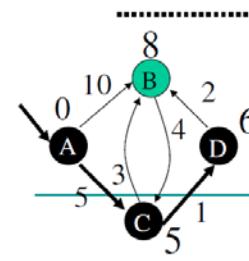
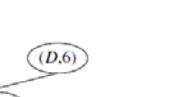
$$d[v] = d[u] + w(u, v)$$



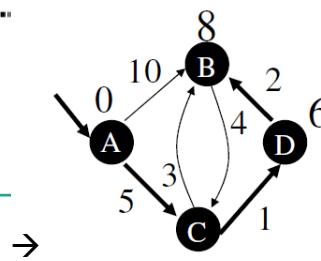
Extract-Min: A  
Decrease-Key: B,C



Extract-Min: C  
Decrease-Key: B,D



Extract-Min: D  
Decrease-Key: B



Extract-Min: B

$Q:$  EMPTY

### Çalışma Zamanı:

Extract-Min:  $|V|$  defa çalıştırılır.  
Decrease-Key:  $|E|$  defa çalıştırılır.

$$T(n,m) = O(n \log n + m \log n)$$

### Ağırlıklı Çizgelerde Dijkstra'nın En Kısa Yol Algoritması:

Tüm  $(u,v) \in E$  için eğer  $w(u,v) = 1$  ise bu durumda Dijkstra'nın algoritması iyileştirilebilir mi ?

Öncelik kuyruğu yerine FIFO kullanılırsa, çalışma zamanı iyileştirilebilir.

### Breadth-first search

```

while  $Q \neq \emptyset$ 
  do  $u \leftarrow \text{DEQUEUE}(Q)$ 
    for each  $v \in \text{Adj}[u]$ 
      do if  $d[v] = \infty$ 
        then  $d[v] \leftarrow d[u] + 1$ 
           $\text{ENQUEUE}(Q, v)$ 

```

Çalışma zamanı bu durumda  $\theta(V + E)$  olur.

### Gevşeme (Relaxation) Teknikleri:

- Çizgedeki her bir düğüm için  $s$ 'den  $v$ 'ye olan en kısa yol  $d[v]$  özelliği içinde tutulur.  $d[v]$  en kısa yol öngörüsü olarak adlandırılmaktadır.
- Başlangıçta, tüm en kısa yol öngörüleri sonsuzdur.
- Algoritma çalıştıkça bu değerler  $s$  ile  $v$  arasındaki gerçek değere  $\delta(s,v)$  yaklaşır.
- Bir kenarın  $(u,v)$  gevşeme işlemi,  $u$ 'ya şu ana kadar bulunmuş olan en kısa yolu  $v$ 'ye genişleterek iyileştirilme ihtimalini test etmektedir.
- Gevşeme adımı, en kısa yol öngörüsünü  $d[v]$  üçgen eşitsizliği ile azaltabilir.

$\text{Relax}(u,v,w)$

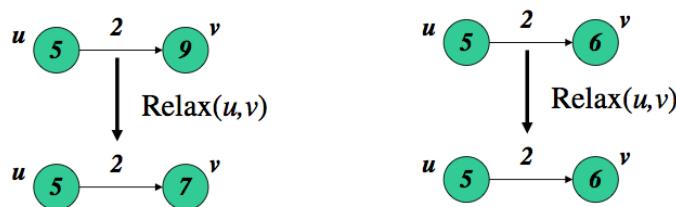
```

if  $d[v] > d[u] + w(u,v)$  the
   $d[v] = d[u] + w(u,v)$ 

```

End

### Örnek:



### Gevşeme özellikleri:

$G=(V,E)$  ağırlıklı bir çizge ve  $w$  kenarlar üzerinde tanımlı bir ağırlık fonksiyonu olsun.

- $(u,v)$ 'yi gevşettikten hemen sonra,  $d[v] \leq d[u] + w(u, v)$
- $G$ 'deki her düğüm  $v$  için :  $d[v] \geq \delta(s,v)$

- $G$ 'de  $s$  ile  $v$  arasında bir yol yok ise  $d[v] = \delta(s, v) = \infty$
- $s$  ile  $v$  arasındaki en kısa yol  $p = <s, \dots, u, v>$  olsun. Eğer  $(u, v)$ 'nin gevşeme işleminden önce herhangi bir zamanda  $d[u] = \delta(s, u)$  ise gevşemeden sonra da  $d[v] = \delta(s, v)$  'dir.

### Bellman-Ford'un en kısa yol Algoritması:

Bu algoritma en kısa yol öngörüsü ile başlar ve bu değer gerçek en kısa yola dönüşür.

SSSP( $G$ )

for each  $v \in V$  do

$$d[v] = \infty$$

$$d[s] = 0$$

for  $i = 1$  to  $|V|$  do

for each edge  $(u, v) \in E$  do

if  $d[v] > d[u] + w(u, v)$  then

$$d[v] = d[u] + w(u, v)$$

for each edge  $(u, v) \in E$  do

if  $d[v] > d[u] + w(u, v)$  then

Output "No Solution!"

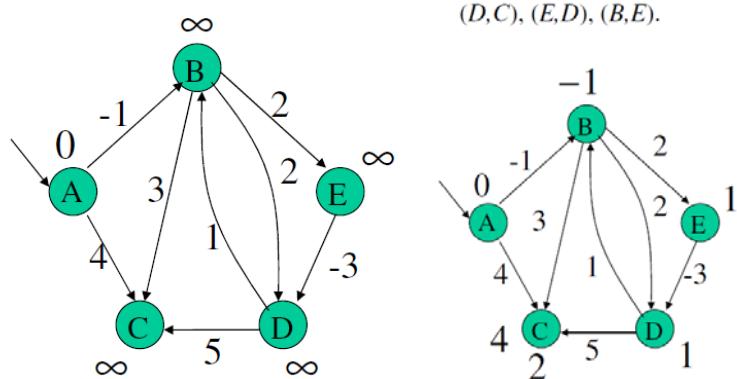
- Algoritmanın başında  $d[]$  değeri sonsuz olarak atanır ve bu değer en kısa yol değerine dönüşür.
- Gevşeme: Her bir kenar  $|V|-1$  defa gevşenmektedir.
- Çizgede, eğer negatif ağırlıklı bir döngü yoksa çözüm bulunur.

**Örnek:**

$$d[A] = 0$$

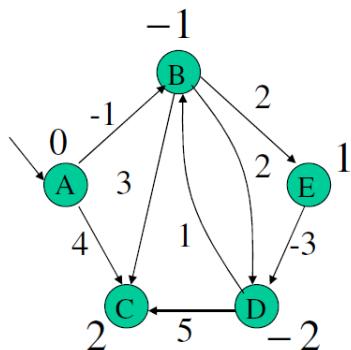
$$d[B] = d[C] = d[D] = d[E] = \infty$$

1. Gevşeme: Kenarlar bu surada alınacak:  
 $(A, B), (A, C), (B, C), (B, D),$   
 $(D, C), (E, D), (B, E).$



2. Gevşeme: Kenarları aynı sırada al:

$(A,B), (A,C), (B,C), (B,D), (D,C), (E,D), (B,E)$ .



**Çalışma Zamanı:**  $O(nm)$ , burada  $n=|V|$  and  $m=|E|$ .

### DAG'lerde En kısa yollar:

DAG (Yönlü ve Döngüsüz Çizgeler)'de en kısa yol bulma algoritmaları çizgelerde negatif döngü olmadığı için daha verimli bir şekilde çalıştırılabilir.

Eğer  $u$ 'dan  $v$ 'ye bir yol var ise, topolojik sıralamada  $u$ ,  $v$ 'den önce gelmektedir. Bu ise, en kısa yolu bulmak için topolojik sıralama yaptıktan sonra düğümleri ele almak manasına gelir.

**DAG-Shortest-Path( $G, w, s$ )**

**Topologically Sort** the vertices of  $G$

**Initialize** the  $d[ ]$  for all the vertices.

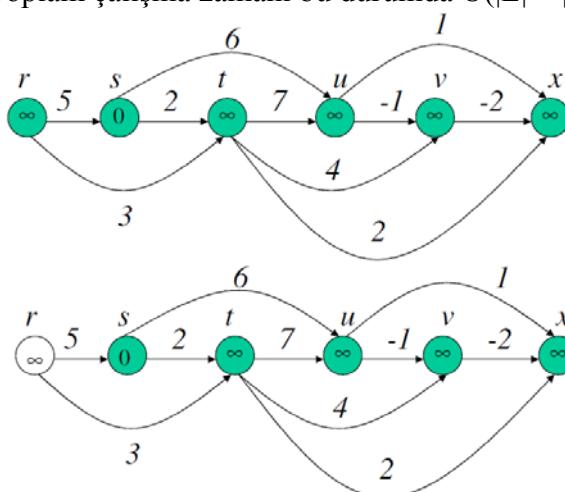
**For** each vertex  $u$  taken in topologically

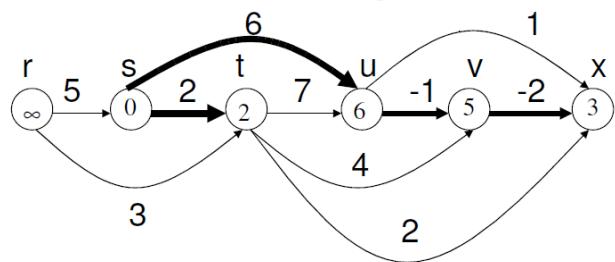
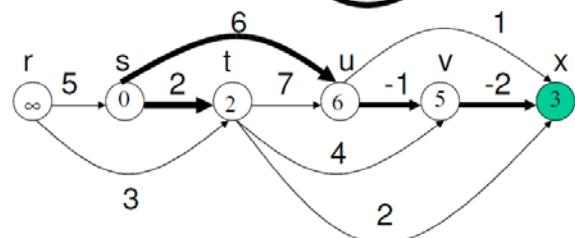
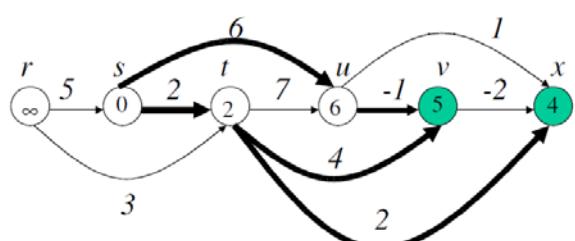
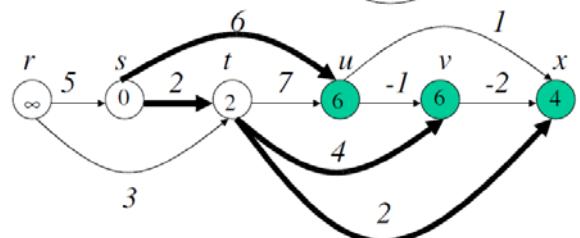
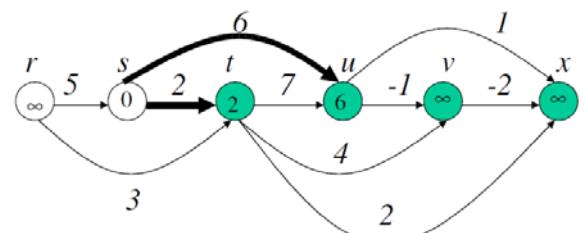
sorted order **do**

**For** each vertex  $v$  in  $Adj[u]$  **do**

**Relax( $u, v, w$ )**

**Çalışma Zamanı:** Her bir düğüm en fazla 1 defa bakılacağı için dışarıdaki For döngüsü  $O(|V|)$  kadar çalıştırılacaktır. İçerideki For döngüsü ise tam olarak  $O(|E|)$  kadar çalıştırılır. Toplam çalışma zamanı bu durumda  $O(|E| + |V|)$ 'dır.





## Tüm düğüm çiftleri arasında en kısa yol bulma:

$G = (V, E)$  ağırlıklı bir çizge ve  $w : E \rightarrow \mathbb{R}$  kenarlar üzerinde tanımlı bir ağırlık fonksiyonu olsun. Amacımız tüm düğüm çiftleri arasındaki en kısa yol değerlerini bulmak  $\delta(u, v)$ . Bellman-Ford algoritmasını her bir düğümden başlayarak çalıştırabiliriz.

- Ancak çalışma zamanı  $O(V^2 E)$  — bu da  $O(V^4)$  eğer ( $E = (V^2)$  ise).

Eğer negatif ağırlıklı döngü yoksa Dijkstra'nın algoritması her düğüm için çalıştırılabilir.

- Bu durumda çalışma zamanı ikili heap'ler ile  $O(VE \lg V)$  — buda  $O(V^3 \lg V)$  eğer ( $E = (V^2)$  ise).

Bu çözümlere karşın çalışma zamanı daha iyi ( $O(V^3 \lg V)$  veya  $O(V^3)$ ) olan algoritmaların bir kısmı aşağıda verilmiştir.

### En kısa yollar ve matris çarpımı:

$G$  çizgesinin komşuluk matrisinde kenar ağırlıkları  $W = (w_{ij})$  verilmiş olsun:

$$w_{ij} = \begin{cases} 0 & \text{eğer } i = j, \\ (i, j) \text{'nin ağırlığı} & \text{eğer } i \neq j, (i, j) \in E, \\ \infty & \text{eğer } i \neq j, (i, j) \notin E. \end{cases}$$

Çıktı,  $D = (d_{ij})$ , matrisidir, burada  $d_{ij} = \delta(i, j)$ .

**Özyinelemeli çözüm:**  $l_{ij}^{(m)}$ i ile j arasındaki en kısa yol ağırlığı olsun ve bu yolda en fazla  $\leq m$  tane kenar bulunsun.

- $m=0$  ise i ile j arasında en fazla  $\leq m$  tane kenarı olan en kısa bir yol vardır ancak ve ancak  $i=j$  ise.

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{eğer } i = j, \\ \infty & \text{eğer } i \neq j. \end{cases}$$

- $m \geq l$  ise

$$\begin{aligned} l_{ij}^{(m)} &= \min \left( l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} \right) \\ &\quad (\text{k, j'nin neslinden gelir}) \\ &= \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} \\ &\quad \text{tüm j'ler için } w_{jj} = 0 \text{ olduğundan.} \end{aligned}$$

- $m=1$  ise  $l_{ij}^{(1)} = w_{ij}$

Çözüm aşağıdan yukarı doğru elde edilebilir. Önce  $L^{(1)}$ , sonra  $L^{(2)}$ ,..vs.., en son  $L^{(n-1)}$  hesaplanır.

$L^{(1)}$  ile başla  $L^{(1)} = W$ , çünkü  $l_{ij}^{(1)} = w_{ij}$ .  
 $L^{(m-1)}$ , den  $L^{(m)}$ ,I hesapla:

```

EXTEND( $L, W, n$ )
create  $L'$ , an  $n \times n$  matrix
for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do  $l'_{ij} \leftarrow \infty$ 
            for  $k \leftarrow 1$  to  $n$ 
                do  $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$ 
return  $L'$ 

```

Her bir  $L^{(m)}$  ‘i hesapla:

```
SLOW-APSP( $W, n$ )
 $L^{(1)} \leftarrow W$ 
for  $m \leftarrow 2$  to  $n - 1$ 
    do  $L^{(m)} \leftarrow \text{EXTEND}(L^{(m-1)}, W, n)$ 
return  $L^{(n-1)}$ 
```

**Çalışma Zamanı:** EXTEND:  $\theta(n^3)$  ve SLOW-APSP: $\theta(n^4)$

EXTEND algoritması matris çarpımı benzeri bir algoritmadır.

$L$	$\rightarrow$	$A$
$W$	$\rightarrow$	$B$
$L'$	$\rightarrow$	$C$
min	$\rightarrow$	+
+	$\rightarrow$	.
$\infty$	$\rightarrow$	0

$n \times n$ ’lik bir C matrisi oluşturur:

```
for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do  $c_{ij} \leftarrow 0$ 
            for  $k \leftarrow 1$  to  $n$ 
                do  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```

EXTEND’ın matris çarpımı olarak görülmesi neden önemlidir ? Çünkü amacımız  $L^{(n-1)}$  ‘i mümkün olduğunda hızlı hesaplamaktır. Ara değerleri  $L^{(1)}, L^{(2)}, L^{(3)}, \dots, L^{(n-2)}$  hesaplamaya gerek yok.

Elimizde A matrisi olduğunu ve  $A^{(n-1)}$ , ‘i hesaplamak istediğimizi kabul edin.

$A, A^{(2)}, A^{(4)}, A^{(8)}, \dots$  hesaplayabiliriz.

Ancak eğer tüm  $m \geq n - 1$ ’ler için  $A^{(m)} = A^{(n-1)}$  olduğunu biliyorsak,  $n - 1$ ’den büyük yada eşit olan en küçük 2’nin üssü değer ( $r$ ) için  $A^{(r)}$  ile amacımıza ulaşabiliriz. ( $r = 2^{\lceil \lg(n-1) \rceil}$ ).

```
FASTER-APSP( $W, n$ )
 $L^{(1)} \leftarrow W$ 
 $m \leftarrow 1$ 
while  $m < n - 1$ 
    do  $L^{(2m)} \leftarrow \text{EXTEND}(L^{(m)}, L^{(m)}, n)$ 
     $m \leftarrow 2m$ 
return  $L^{(m)}$ 
```

Çalışma Zamanı:  $\theta(n^3 \lg n)$ .

## Floyd-Warshall algoritması

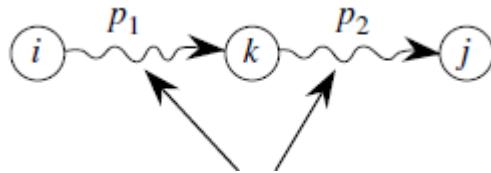
Farklı bir dinamik programlama metodu.

$p = \langle v_1, v_2, \dots, v_l \rangle$  yolu için  $p$ ’nin ara düğümü  $v_1$  ve  $v_l$  dışındaki herhangi bir düğümdür.

$d_{ij}^{(k)}$ , tüm ara düğümleri  $\{1, 2, \dots, k\}$  kümesi içinden olan i ile j arasındaki en kısa yol ağırlığı olsun.

Tüm ara düğümleri  $\{1, 2, \dots, k\}$  kümesi içinden olan i ile j arasındaki en kısa yol ( $p$ ) ‘yi göz önüne alın.

- Eğer  $k$  ara düğüm değilse, tüm ara düğümler  $\{1, 2, \dots, k-1\}$  kümesi içindedir.
- Eğer  $k$  ara düğüm ise,



tüm ara düğümler  $\{1, 2, \dots, k-1\}$  kümesi içindedir.

### Özyinelemeli formulasyon:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{eğer } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{eğer } k \geq 1. \end{cases}$$

$d_{ij}^{(0)} = w_{ij}$  olmalıdır çünkü  $\leq 1$  kenara sahip ara düğümler olamaz.

Tüm düğümler  $\leq n$  numaralı olduğu için  $D^{(n)} = (d_{ij}^{(n)})$  olmasını istiyoruz.

Algoritma, artan  $k$  değerlerine göre hesaplar:

**FLOYD-WARSHALL( $W, n$ )**

$D^{(0)} \leftarrow W$

**for**  $k \leftarrow 1$  **to**  $n$

**do for**  $i \leftarrow 1$  **to**  $n$

**do for**  $j \leftarrow 1$  **to**  $n$

**do**  $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

**return**  $D^{(n)}$

**Çalışma Zamanı:**  $\theta(n^3)$ .

### Transitive closure:

Verilen bir yönlü  $G=(V, E)$  çizgesi için  $G^*=(V,E^*)$  çizgesini hesapla.

- $E^* = \{(i, j) : G$ 'de  $i$  ile  $j$  arasında bir yol var ise}

Bunun için her bir kenara 1 ağırlığı verip FLOYD-WARSHALL algoritmasını çalıştırabiliriz.

- Eğer  $d_{ij} < n$  ise  $i$  ile  $j$  arasında bir yol vardır.
- Diğer durumda,  $d_{ij} = \infty$  ve bir yol yoktur.

### Daha kolay bir yol:

FLOYD-WARSHALL algoritmasındaki işleçleri değiştirebiliriz.

- Ağırlıksız komşuşuk matrisi kullan
- $\min \rightarrow V$  (veya)
- $+$   $\rightarrow \wedge$  (veya)

- $t_{ij}^{(k)} = \begin{cases} 1 & \text{tüm ara düğümleri } \{1, 2, \dots, k\} \text{ kümelerinden olan } i \text{ ile } j \text{ arasında bir yol var ise} \\ 0 & \text{diğer durumlarda.} \end{cases}$
- $t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ ve } (i, j) \notin E, \\ 1 & \text{if } i = j \text{ veya } (i, j) \in E. \end{cases}$
- $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$

**TRANSITIVE-CLOSURE( $E, n$ )**

```

for  $i \leftarrow 1$  to  $n$ 
  do for  $j \leftarrow 1$  to  $n$ 
    do if  $i = j$  or  $(i, j) \in E[G]$ 
      then  $t_{ij}^{(0)} \leftarrow 1$ 
      else  $t_{ij}^{(0)} \leftarrow 0$ 
for  $k \leftarrow 1$  to  $n$ 
  do for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
      do  $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
return  $T^{(n)}$ 
```

**Çalışma Zamanı:**  $\theta(n^3)$ , fakat FLOYD-WARSHALL'a göre daha kolay işlemleri içermektedir.