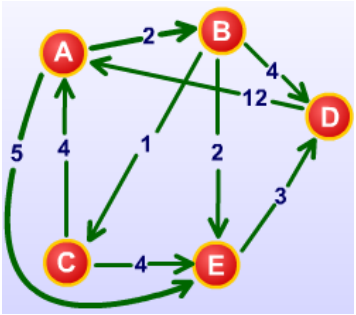
ÇİZGİ KÜMELERİ (GRAPHS)

GRAFLAR

- **Tanım**
  - Yönlendirilmiş ve yönlendirilmemiş graflar
  - Ağırlıklı graflar
- **Gösterim**
  - Komşuluk Matrisi
  - Komşuluk Listesi
- **Dolaşma Algoritmaları**
  - BFS (Breath First Search)
  - DFS (Depth-First Search)

GRAFLAR



- Graf, matematiksel anlamda, düğümlerden ve bu düğümler arasındaki ilişkiyi gösteren kenarlardan oluşan bir kümedir. Mantıksal ilişki, düğüm ile düğüm veya düğüm ile kenar arasında kurulur.
  - Bağlantılı listeler ve ağaçlar grafların özel örneklerindendir.
- Fizik, Kimya gibi temel bilimlerde ve mühendislik uygulamalarında ve tıp biliminde pek çok problemin çözümü ve modellenmesi graflara dayandırılarak yapılmaktadır.

GRAFLAR-Uygulama alanları

- **Elektronik devreler**
  - Baskı devre kartları (PCB), Entegre devreler
- **Ulaşım ağları**
  - Otoyol ağı, Havayolu ağı
- **Bilgisayar ağları**
  - Lokal alan ağları ,İnternet
- **Veritabanları**
  - Varlık-ilişki (Entity-relationship) diyagramı

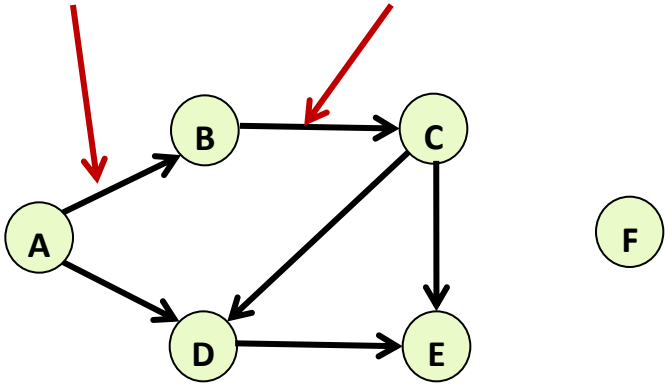
GRAFLAR

- Bir G grafı D ile gösterilen düğümlerden (node veya vertex) ve K ile gösterilen kenarlardan (Edge) oluşur.
  - Her kenar iki düğümü birleştirirerek iki bilgi (Düğüm) arasındaki ilişkiyi gösterir ve (u,v) şeklinde ifade edilir. (u,v) iki düğümü göstermektedir.

- Bir graf üzerinde n tane düğüm ve m tane kenar varsa, matematiksel gösterilimi, düğümler ve kenarlar kümesinden elamanların ilişkilendirilmesiyle yapılır:
  - $D=\{d_0, d_1, d_2 \dots d_{n-1}, d_n\}$ Düğümler kümesi
  - $K=\{k_0, k_1, k_2 \dots k_{m-1}, k_m\}$ Kenarlar kümesi
  - $G=(D, K)$ Graf

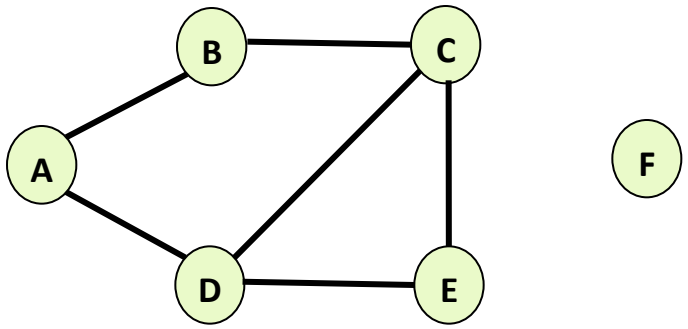
Graflar

- G = (D, K) grafı aşağıda verilmiştir.
  - D = {A, B, C, D, E, F}
  - K = {(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)}



Graflar – Tanımlar

- Komşu(Adjacent): Bir G grafı üzerindeki d_i ve d_j adlı iki düğüm, kenarlar kümesinde bulunan bir kenarla ilişkilendiriliyorsa bu iki düğüm birbirine komşu (adjacent, neighbor) düğümlerdir; k={d_i, d_j} şeklinde gösterilir ve k kenarı hem d_i hem de d_j düğümleriyle bitişiktir (incident) denilir.
- Diğer bir deyişle k kenarı d_i ve d_j düğümlerini birbirine bağlar veya d_i ve d_j düğümleri k kenarının uç noktalarıdır denilir.
  - (A, B) komşudur.
  - (B, D), (C, F) komşu değildir.



Graflar – Tanımlar

- Komşuluk ve Bitişiklik: Bir G grafı komşuluk ilişkisiyle gösteriliyorsa $G_{dd}=\{(d_i, d_j)...\}$, bitişiklik ilişkisiyle gösteriliyorsa $G_{dk}=\{(d_i, k_j)...\}$ şeklinde yazılır. (G_{dd}: $G_{düğümdüğüm}$, G_{dk}: $G_{düğümkenar}$)
  - Örneğin, şekil a) 2-düğümlü basit graf $G_{dd}=\{(d_0, d_1)\}$ veya $G_{dk}=\{(d_0, k_0), (d_1, k_0)\}$ şeklinde yazılabilir;
  - b) verilen 4-düğümlü basit grafta $G_{dd}=\{(d_0, d_0), (d_0, d_1), (d_0, d_2), (d_0, d_3), (d_1, d_2), (d_2, d_3)\}$ veya $G_{dk}=\{(d_0, k_0), (d_0, k_1), (d_0, k_2), (d_0, k_3), (d_1, k_1), (d_1, k_4), (d_2, k_2), (d_2, k_4), (d_2, k_5), (d_3, k_3), (d_3, k_5)\}$ şeklinde yazılır.

Graflar – Tanımlar

- **Yönlendirilmiş Graf (Directed Graphs):** Bir G grafı üzerindeki kenarlar bağlantının nereden başlayıp nerede sonlandığını belirten yön bilgisine sahip ise yönlü-graf veya yönlendirilmiş graf (directed graf) olarak adlandırılır.

- Yönlü graflar, matematiksel olarak gösterilirken her bir ilişki oval parantezle değil de **<>** karakter çiftiyle gösterilir.

- **Yönlendirilmemiş Graf (Undirected Graphs)**
  - Hiçbir kenarı yönlendirilmemiş graftır.

Graflar – Tanımlar

- **Yönlendirilmiş Kenar (Directed Edge)**
  - Sıralı kenar çiftleri ile ifade edilir.
    - (u, v) ile (v, u) aynı değildir.
  - İlk kenar orijin ikinci kenar ise hedef olarak adlandırılır.

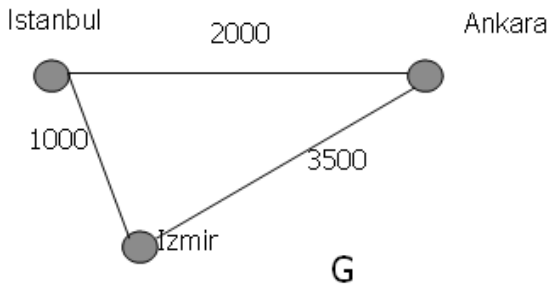
- **Yönlendirilmemiş Kenar (Undirected Edge)**
  - Sırasız kenar çiftleri ile ifade edilir.
    - (u, v) ile (v, u) aynı şeyi ifade ederler.

Graflar – Tanımlar

- **Ağırlıklı Graf(Weighted Graphs) :**

- Graf kenarları üzerinde ağırlıkları olabilir. Eğer kenarlar üzerinde ağırlıklar varsa bu tür graflara ağırlıklı/maliyetli graf denir. Eğer tüm kenarların maliyeti 1 veya birbirine eşitse maliyetli graf olarak adlandırılmaz; yön bilgisi de yoksa basit graf olarak adlandırılır.

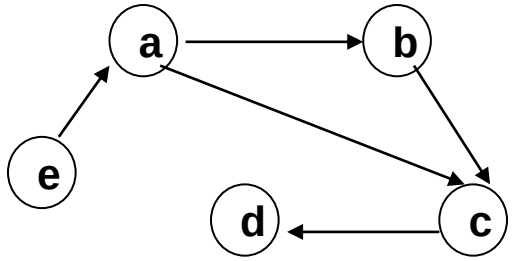
  - Ağırlık uygulamadan uygulamaya değişir.
    - Şehirler arasındaki uzaklık.
    - Routerler arası bant genişliği
      - Router: Yönlendirici



Graflar – Tanımlar

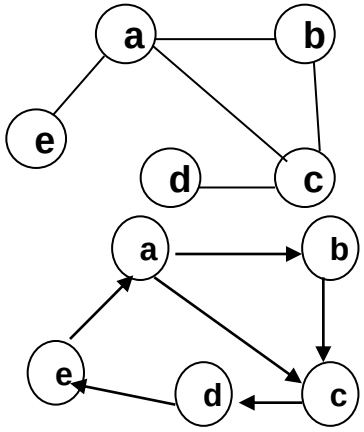
- **Yol (path) :** Bir çizgi kümesinde yol, düğümlerin sırasıdır. Kenar uzunluğu, yolun uzunluğunu verir.
  - $V_1, V_2, V_3 \ldots\ldots\ldots V_N$ ise yol $N - 1$ dir.
  - Eğer yol kenar içermiyorsa o yolun uzunluğu sıfır'dır.
  - izmir'den Ankara'ya doğrudan veya İstanbul'dan geçerek gidilebilir (İzmir- İstanbul- Ankara).

- **Basit Yol (Simple Path) :** Bir yolda ilk ve son düğümler dışında tekrarlanan düğüm yoksa basit yol diye adlandırılır.
  - eabcd basit yol'dur fakat eabcabcd veya abcabca basit yol değildir.

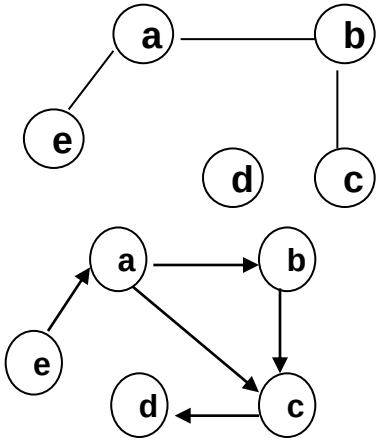


Graflar – Tanımlar

- **Uzunluk :** Bir yol üzerindeki kenarların uzunlukları toplamı o yolun uzunluğudur.
- **Bağlı veya Bağlı olmayan Çizge(Connected Graph):**
- Eğer bir graftaki tüm düğümler arasında en azından bir yol varsa bağlı graftır. Eğer bir grafta herhangi iki düğüm arasında yol bulunmuyorsa bağlı olmayan graftır.
- A) Bağlı Graf



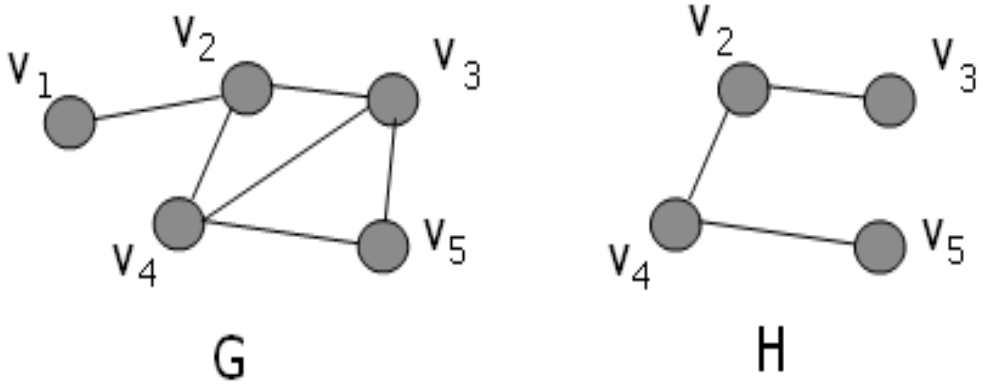
B) Bağlı olmayan Graf



Graflar – Tanımlar

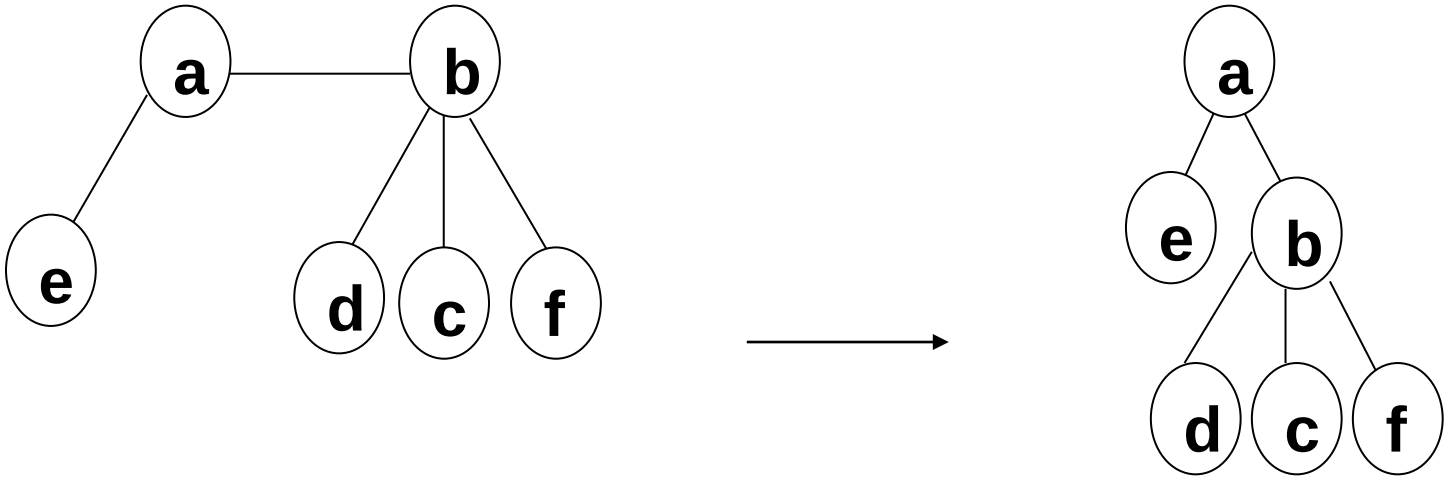
- **Alt Graf (Subgraph) :** H çizgesinin köşe ve kenarları G çizgesinin köşe ve kenarlarının alt kümesi ise; H çizgesi G çizgesinin alt çizgesidir (subgraph).

- G (V, E) şeklinde gösterilen bir grafın, alt grafı H(U, F) ise U alt küme V ve F alt küme E olur.



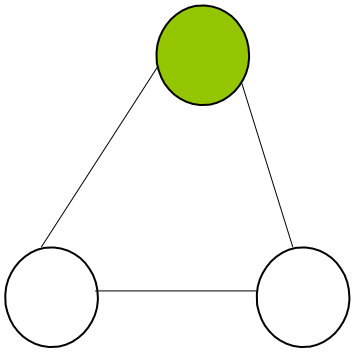
Graflar – Tanımlar

- **Ağaçlar(Trees) :** Ağaçlar özel bir çizgi kümesidir. Eğer direk olmayan bir çizgi kümesi devirli (cycle) değilse ve de bağlıysa (connected) ağaç olarak adlandırılır.

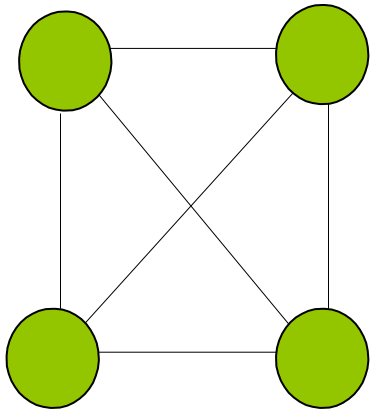


Graflar – Tanımlar

- **Komple Çizge(Complete Graph) :** Eğer bir graftaki her iki node arasında bir kenar varsa komple graftır.



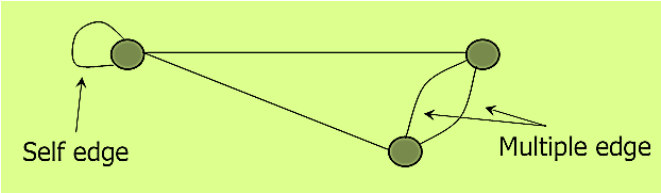
- 3 node ile komple graf



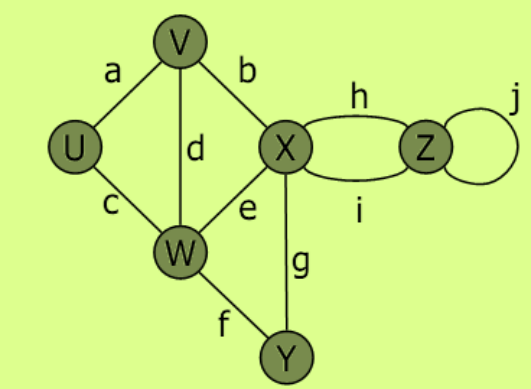
4 node ile komple graf

Graflar – Tanımlar

- **Multigraf:** Multigraf iki node arasında birden fazla kenara sahip olan veya bir node'un kendi kendisini gösteren kenara sahip olan graftır.



- Örnek
- a, b ve d kenarları V node'unun kenar bağlantılarıdır.
- X node'unun derecesi 5' tir.
- h ve i çoklu (multiple) kenarlardır.
- j kendi kendisine döngüdür (self loop)



Graflar – Tanımlar

- **Düğüm Derecesi (Node Degree):**

- Düğüme bağlı toplam uç sayısıdır; çevrimli kenarlar aynı düğüme hem çıkış hem de giriş yaptığı için dereceyi iki arttırır.

- Yönlü graflarda, düğüm derecesi giriş derecesi (input degree) ve çıkış derecesi (output degree) olarak ayrı ayrı belirtilir.

Graflar – Tanımlar

- **Komşuluk Matrisi (Adjacency Matrice):**
  Düğümlerden düğümlere olan bağlantıyı gösteren bir kare matrisdir; komşuluk matrisinin elemanları ki değerlerinden oluşur. Komşuluk matrisi G_{dd}'nin matrisel şekilde gösterilmesinden oluşur. Eğer komşuluk matrisi $G_{dd}=[a_{ij}]$ ise,

- yönlü-maliyetsiz graflar için; $a_{ij} = \begin{cases} 1, \text{Eğer } (d_i, d_j) \in K \text{ ise} \\ 0, \text{Diğer durumlarda} \end{cases}$
- olur.

- basit (yönsüz-maliyetsiz) graflar için ise,
- olur.

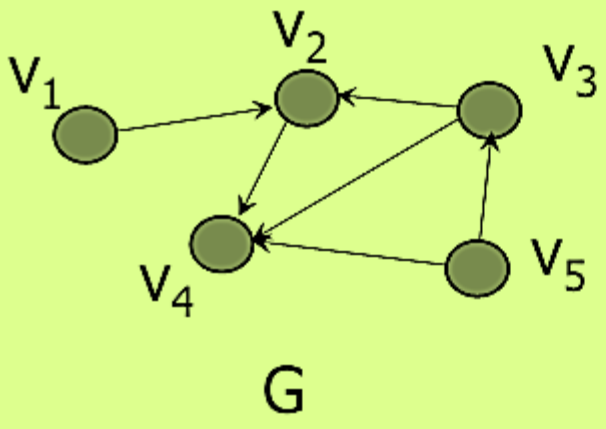
$$a_{ij} = \begin{cases} 1, \text{Eğer } (d_i, d_j) \in K \text{ ise veya } (d_j, d_i) \\ 0, \text{Diğer durumlarda} \end{cases}$$

Komşuluk Matrisi

- Yönlendirilmiş graf için komşu matrisi

Matris[i][j] = 1 if $(v_i, v_j) \in E$

0 if $(v_i, v_j) \notin E$



G

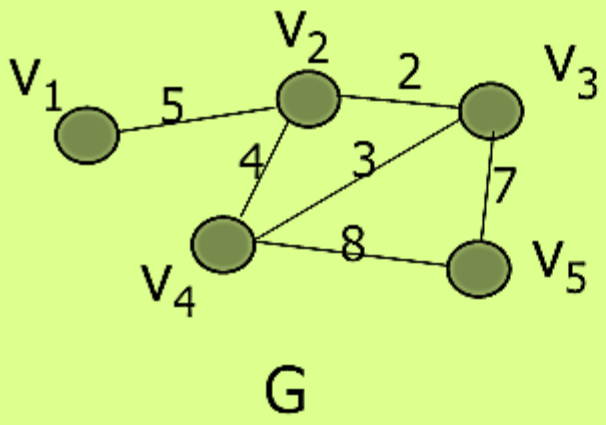
		1 (v_1)	2 (v_2)	3 (v_3)	4 (v_4)	5 (v_5)
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

Komşuluk Matrisi

- Ağırlıklandırılmış ancak yönlendirilmemiş graf için komşu matrisi

Matris[i][j] = $w(v_i, v_j)$ if $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$

∞ otherwise



		1 (v_1)	2 (v_2)	3 (v_3)	4 (v_4)	5 (v_5)
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	∞	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

Graflar – Tanımlar

- **Bitişiklik Matrisi (Incedence Matrice):**
- Düğümlerle kenarlar arasındaki bağlantı/bitişiklik ilişkisini gösteren bir matrisdir; matrisin satır sayısı düğüm, sütun sayısı kenar sayısına kadar olur. Bitişiklik matrisi G_{dk}'nin matrisel şekilde gösterilmesinden oluşur. Eğer bitişiklik matrisi $G_{dk}=[m_{ij}]$ ise, maliyetsiz graflar için,

$$m_{ij} = \begin{cases} 1, \text{ Eğer } (d_i, d_j) \text{ bağlantı var ise} \\ 0, \text{ Diğer durumlarda} \end{cases}$$

- olur.

Graflar – Tanımlar

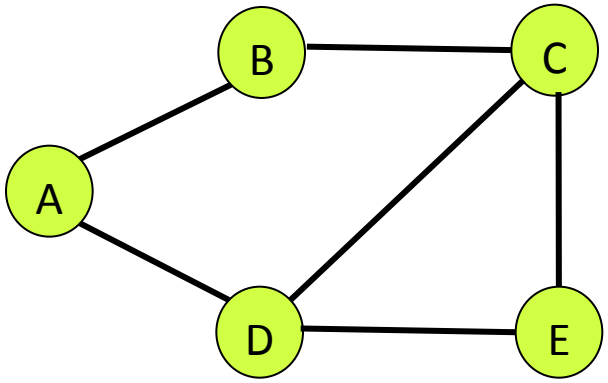
- **Örnek:** Basit grafın komşuluk ve bitişiklik matrisi çıkarılması
- Aşağıda a)'da verilen grafın komşuluk ve bitişiklik matrislerini ayrı ayrı çıkarınız; hangisi simetriktir? Neden?

	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	1	1
D	1	1	1	0

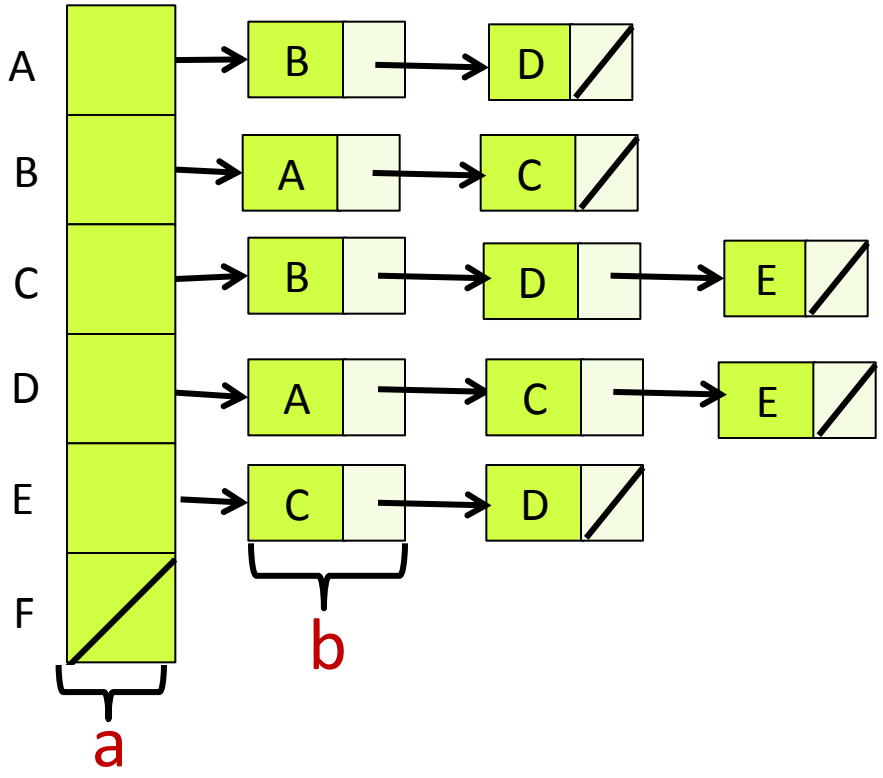
	k_0	k_1	k_2	k_3	k_4	k_5
A	1	1	0	0	0	0
B	0	0	1	1	0	0
C	1	0	1	0	1	1
D	0	1	0	1	0	1

- Komşuluk matrisi simetriktir. Yönsüz graflarda, her iki yönde bağlantı sağlanmaktadır.

Komşuluk Listesi (Dizi Bağlantılı Liste) Gösterimi



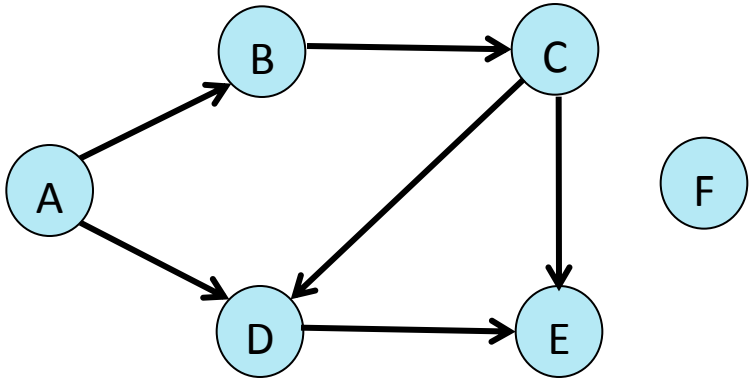
F

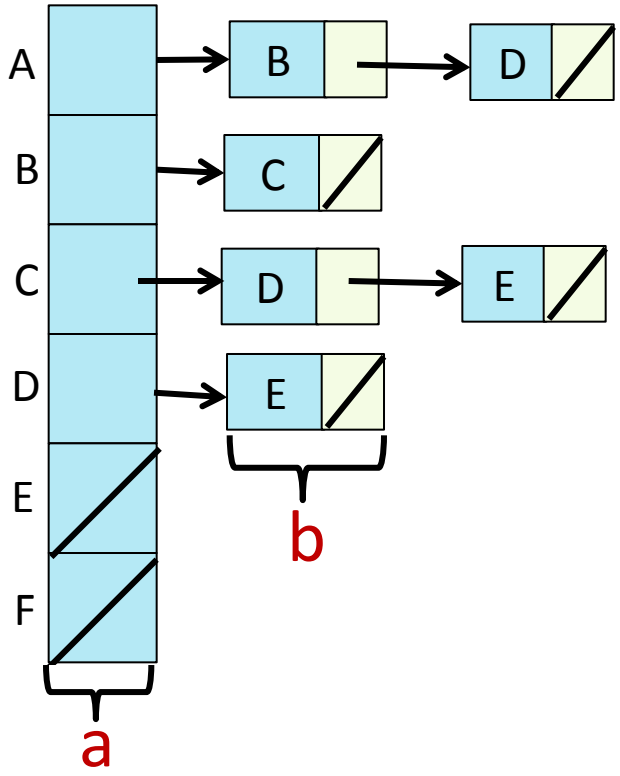


Yer? $n*a + 2*k*b = O(n+2k)$

Komşuluk Listesi Gösterimi

- Komşuluk Listesi (Yönlendirilmiş Graflar)

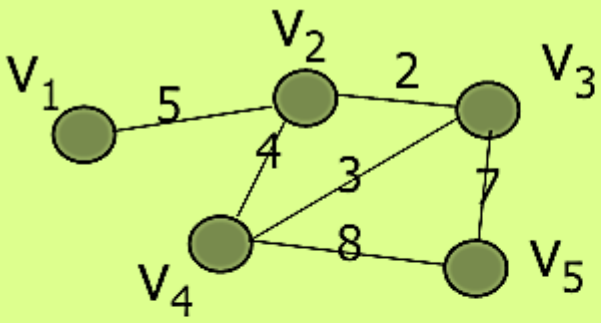




Yer? $n*a + k*b = O(n+k)$

Komşuluk Listesi Gösterimi

- Yönlendirilmiş ve ağırlıklandırılmış graf için komşu listesi



G

1	v_1	$\rightarrow v_2(5)$
2	v_2	$\rightarrow v_1(5) \rightarrow v_3(2) \rightarrow v_4(4)$
3	v_3	$\rightarrow v_2(2) \rightarrow v_4(3) \rightarrow v_5(7)$
4	v_4	$\rightarrow v_2(4) \rightarrow v_3(3) \rightarrow v_5(8)$
5	v_5	$\rightarrow v_3(7) \rightarrow v_4(8)$

Komşu Matrisi-Komşu Listesi

- **Avantajları-dezavantajları;**

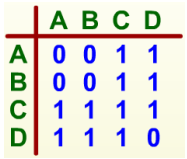
- **Komşu matrisi**
  - Çok fazla alana ihtiyaç duyar. Daha az hafızaya ihtiyaç duyulması için sparse (seyrek matris) matris tekniklerinin kullanılması gerekir.
  - Herhangi iki node'un komşu olup olmadığına çok kısa sürede karar verilebilir.

- **Komşu listesi**
  - Bir node'un tüm komşularına hızlı bir şekilde ulaşılır.
  - Daha az alana ihtiyaç duyar.
  - Oluşturulması matrise göre daha zor olabilir.

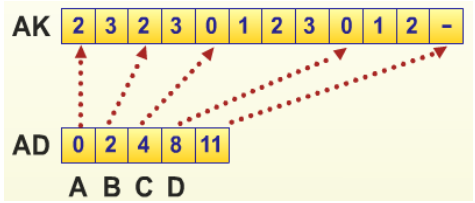
Not: Sparse matris; m*n boyutlu matriste değer bulunan hücreleri x boyutlu matriste saklayarak hafızadan yer kazanmak için kullanılan yöntemin adı.

Grafların Bellek Üzerinde Tutulması

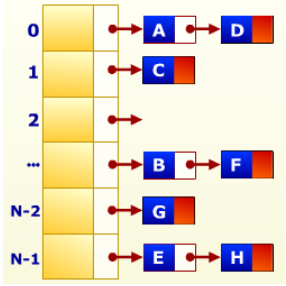
- Matris üzerinde



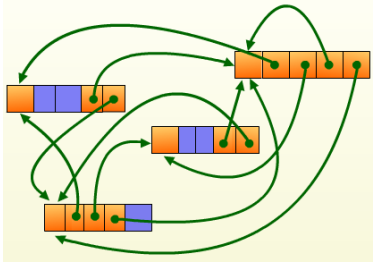
- İki-Dizi Üzerinde



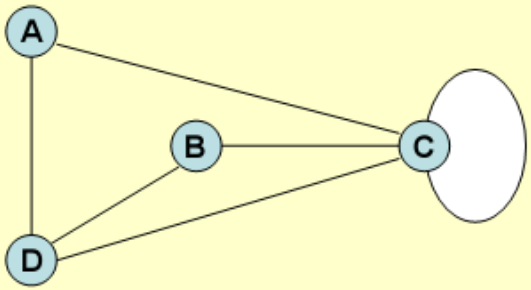
- Dizili Bağlantılı liste ile



- Bağlantılı Liste ile



Matris Üzerinde



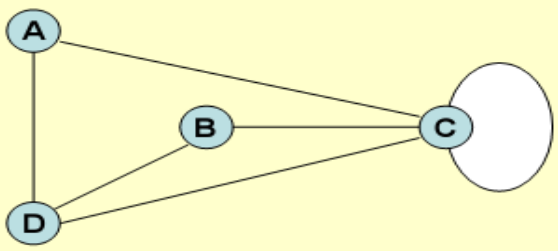
A	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	1	1
D	1	1	1	0

$$A=\begin{vmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{vmatrix}$$

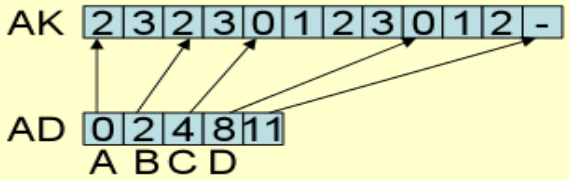
- int A[4][4]={{0,0,1,1}, {0,0,1,1}, {1,1,1,1}, {1,1,1,0}}
- int baglantivarmi (int durum1, int durum2)
- {
- if(A[durum1][durum2]!=0)
- return 0;
- else
- return 1;
- }

İki-Dizi Üzerinde

- Komşular AK, İndisler AD yi oluşturuyor

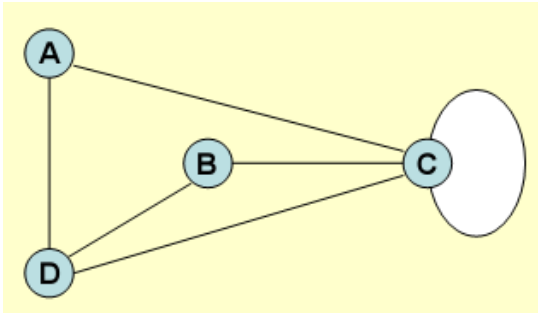


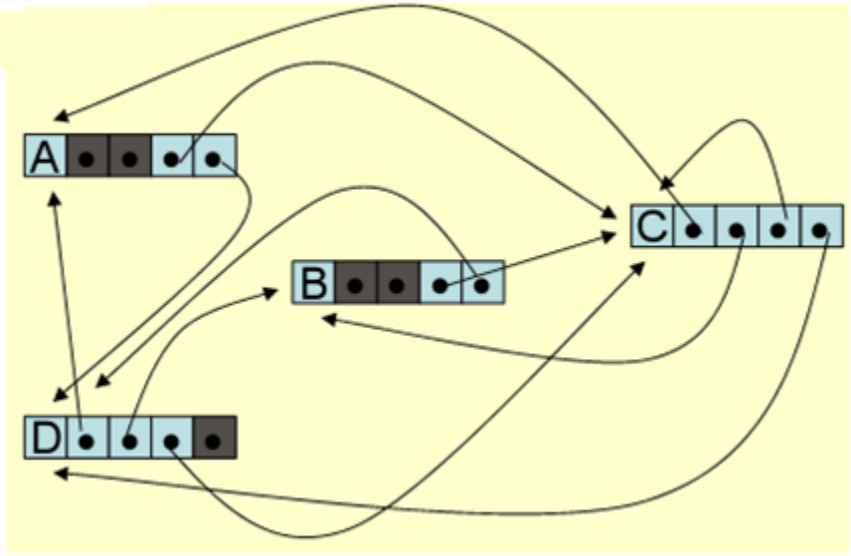
A	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	1	1
D	1	1	1	0



- A=0, B=1, C=2, D=3 numaralanmış
- A, 2 ve 3 komşu başlangıç indisi 0 da 2, indis 1 de 3
- B, 2 ve 3 komşu başlangıç indisi 2 de 2 indis 3 te 3
- C, 0, 1, 2 ve 3 komşu başlangıç indisi 4 de 0, 5 te 1, 6 da 2, 7 de 3
- D, 0, 1 ve 2 ye komşu başlangıç indisi 8 de 0 , 9 da 1, 10 da 2

Bağlantılı Liste Üzerinde

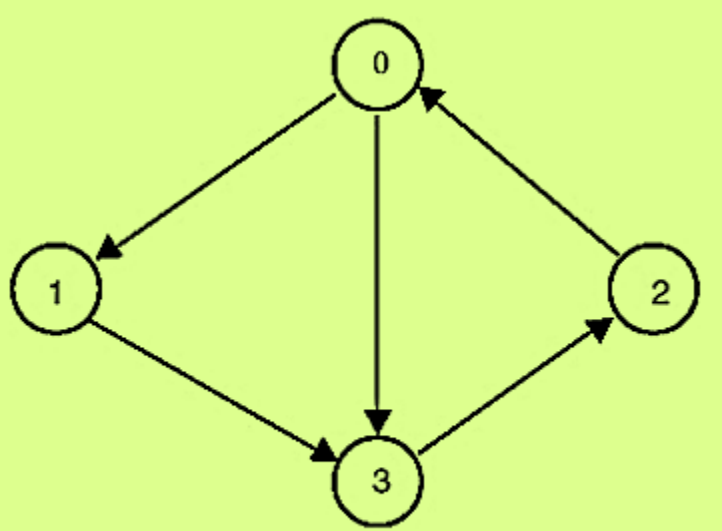




- struct grafDrum{
- verituru dugumadi;
- struct graftdrum *bag[4];
- };

Örnekler

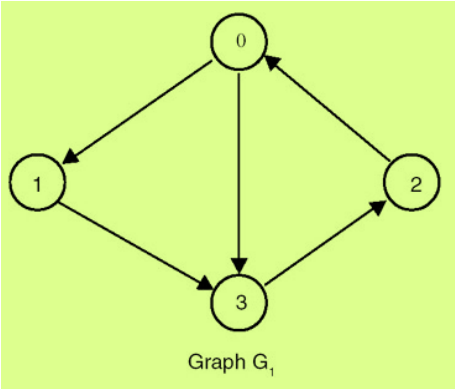
- Indegree (girenler) ve outdegree (çıkanlar) matris üzerinde gösterimi



Graph G_1

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

Örnekler



Graph G_1

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

- Indegree ve outdegree hesaplanması

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
/* Bitişiklik matrisine ait fonksiyonun tanımlanması*/
void buildadjm(int adj[][MAX], int n)   {
 int i,j;
for(i=0;i<n;i++)
for(j=0;j<n;j++)        {
printf("%d ile %d arasında kenar var ise 1 yoksa 0 giriniz\n",i,j);
scanf("%d",&adj[i][j]);        }   }
```

Örnekler

- /* Outdegree ye ait düğümlerin hesaplandığı fonksiyon*/

- int outdegree(int adj[][MAX],int x,int n) {
- int i, count =0;
- for(i=0;i<n;i++)
- if(adj[x][i] ==1) count++; return(count);
- }

Örnekler

- /* Indegree ye ait düğümlerin hesaplandığı fonksiyon */

- int indegree(int adj[][MAX],int x,int n)
- {
- int i, count =0;
- for(i=0;i<n;i++)
- if(adj[i][x] ==1) count++; return(count);
- }

Örnekler

```
void main()    {
int adj[MAX][MAX],node,n,i;
printf("Graph için maksimum düğüm sayısını giriniz (Max= %d):",MAX);

scanf("%d",&n);

buildadjm(adj,n);
for(i=0;i<n;i++)
{
printf("\n%d Indegree düğüm sayısı %d dır ",i,indegree(adj,i,n));
printf("\n%d Outdegree düğüm sayısı %d dır ",i,outdegree(adj,i,n));   }
}
```

Graf Üzerinde Dolaşma

- Graf üzerinde dolaşma grafın düğümleri ve kenarları üzerinde istenen bir işi yapacak veya bir problemi çözecek biçimde hareket etmektir.

- Graf üzerinde dolaşma yapan birçok yaklaşım yöntemi vardır; en önemli iki tanesi, kısaca, DFS (Depth First Search) ve BFS (Breadth First Search) olarak adlandırılmıştır ve graf üzerine geliştirilen algoritmaların birçoğu bu yaklaşım yöntemlerine dayanmaktadır denilebilir.

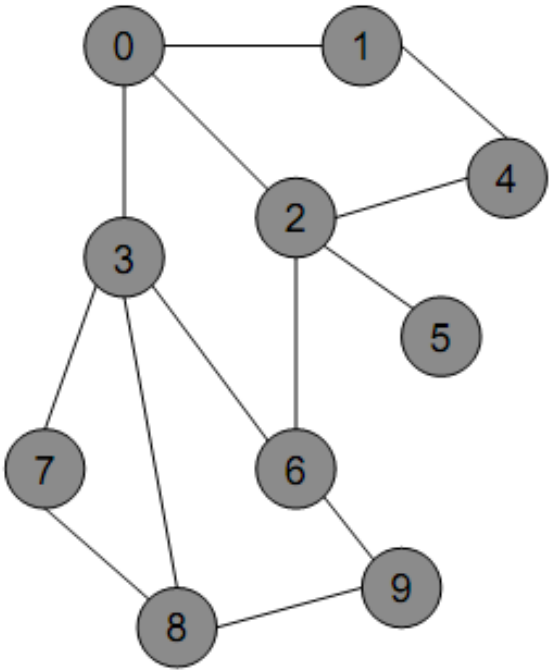
Graf Üzerinde Dolaşma

- **DFS Yöntemi**
- DFS (Depth First Search), graf üzerinde dolaşma yöntemlerinden birisidir; **önce derinlik araması** olarak adlandırılabilir; başlangıç düğümünün bir kenarından başlayıp o kenar üzerinden gidilebilecek en uzak (derin) düğüme kadar sürdürülür.

Graf Üzerinde Dolaşma

- Depth First Arama İşlem Adımları
  - Önce bir başlangıç node'u seçilir ve ziyaret edilir.
  - Seçilen node'un bir komşusu seçilir ve ziyaret edilir.
  - 2.adım ziyaret edecek komşu kalmayıncaya kadar tekrar edilir.
  - Komşu kalmadığında tekrar geri dönülür ve önceki ziyaret edilmiş node'lar için adım 2 ve 3 tekrar edilir.

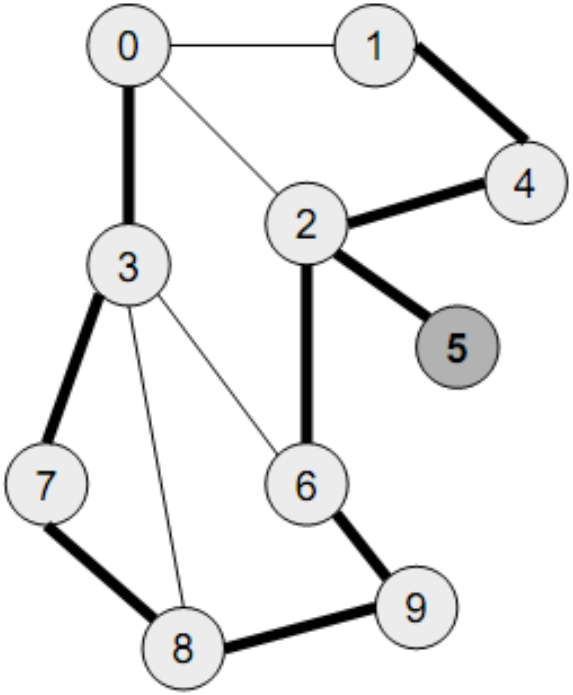
Graf Üzerinde Dolaşma

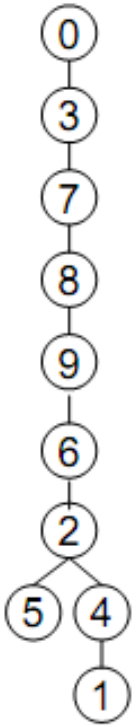



```
1.  s node'unu seç
2.  visit s
    // örn. ekrana yaz
3.  for each edge <s, U>
    // U komşu node
4.      if U is not visit
5.          DFS(G, U)
```

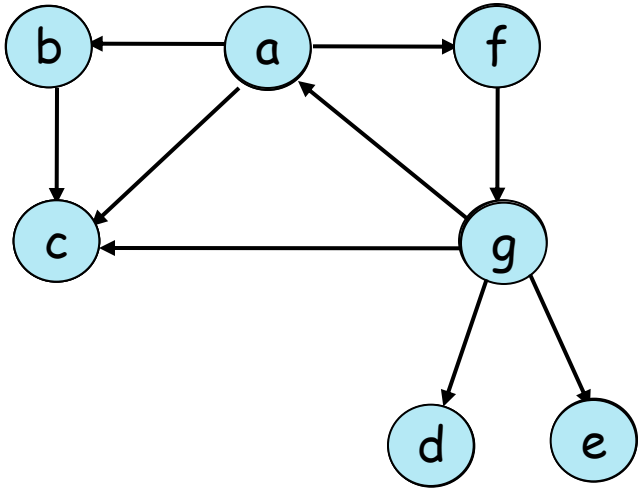
Graf Üzerinde Dolaşma

- **Depth first arama**

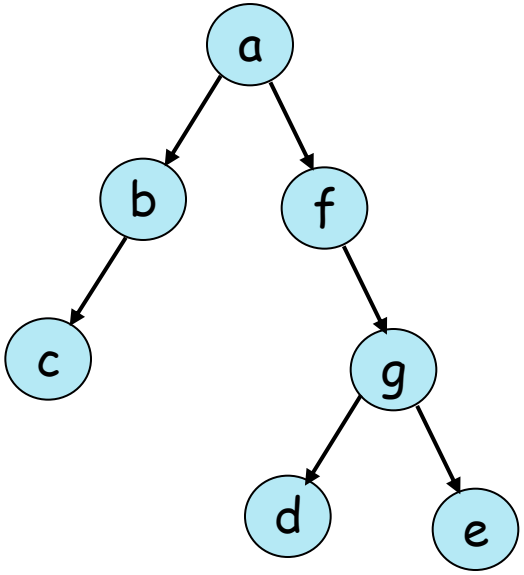




DFS – Örnek



DFS(a) ağacı



Graf Üzerinde Dolaşma

- **BFS Yöntemi :**
- BFS (Breadth First Search), önce genişlik araması olarak adlandırılabilir. Bu yöntemin DFS'den farkı, dolaşmaya, başlangıç düğümünün bir kenarı ayrıtı üzerinden en uzağa gidilmesiyle değil de, başlangıç düğümünden gidilebilecek tüm komşu düğümlere gidilmesiyle başlanır.

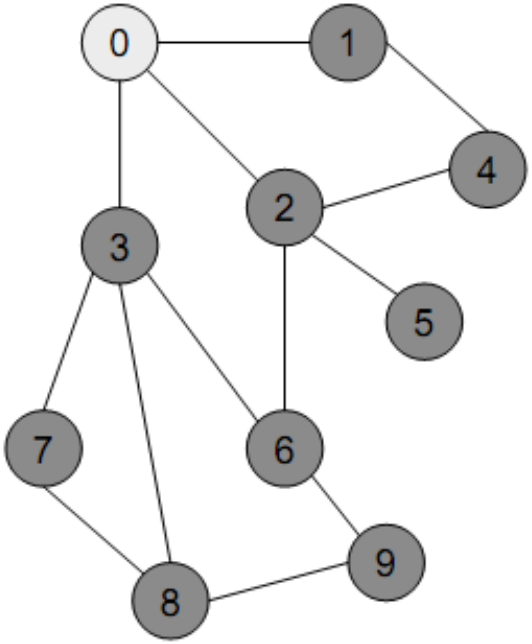
- BFS yöntemine göre graf üzerinde dolaşma, graf üzerinde dolaşarak işlem yapan diğer birçok algoritmaya esin kaynağı olmuştur denilebilir. Örneğin, kenar maliyetler yoksa veya eşitse, BFS yöntemi en kısa yol algoritması gibidir; bir düğümden herbir düğüme olan en kısa yolları bulur denilebilir.

Graf Üzerinde Dolaşma

- Breadth First Arama İşlem Adımları
  - Breadth first arama ağaçlardaki level order aramaya benzer.
  - Seçilen node'un tüm komşuları sırayla seçilir ve ziyaret edilir.
  - Her komşu queue içerisine atılır.
  - Komşu kalmadığında Queue içerisindeki ilk node alınır ve
  - 2.adıma gidilir.

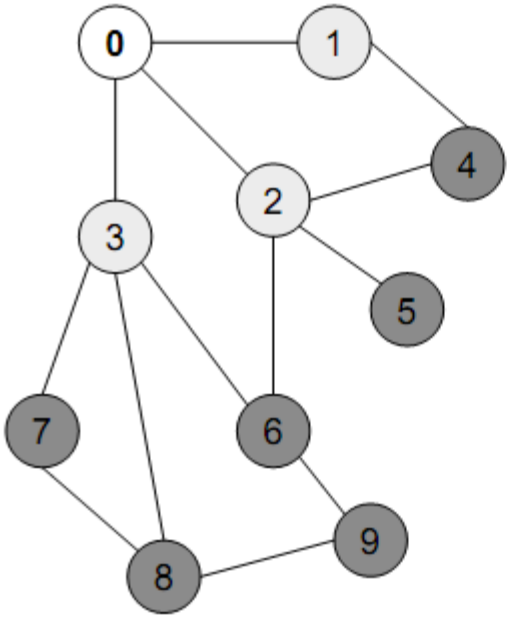
Graf Üzerinde Dolaşma

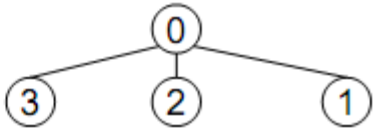
- **Breadth first arama**



Graf Üzerinde Dolaşma

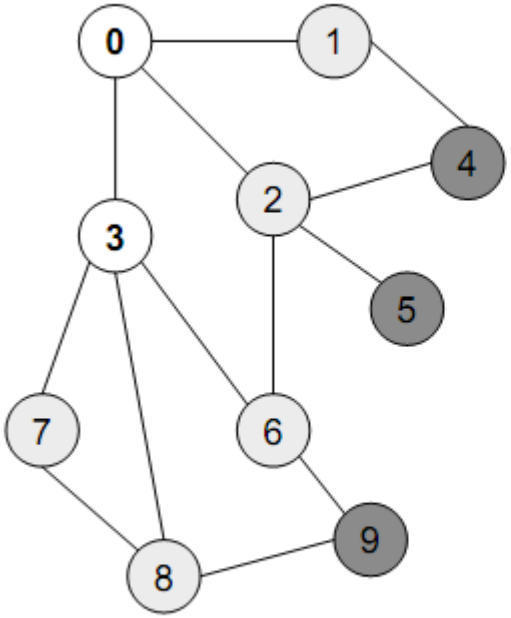
- **Breadth first arama**

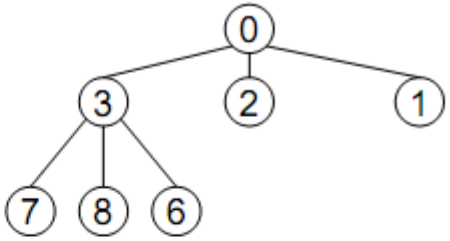




Graf Üzerinde Dolaşma

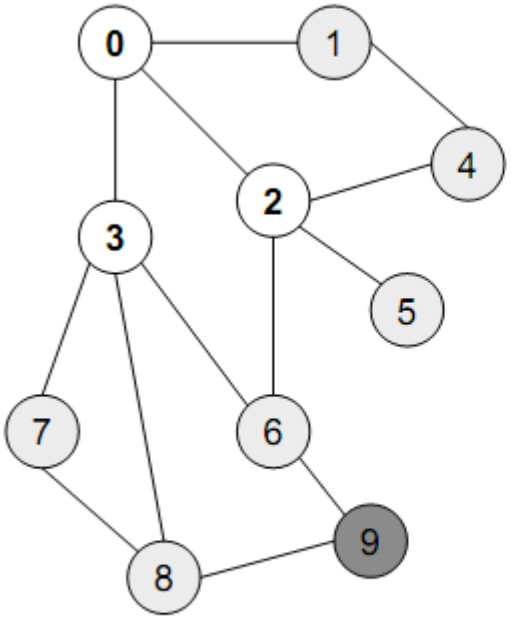
- **Breadth first arama**

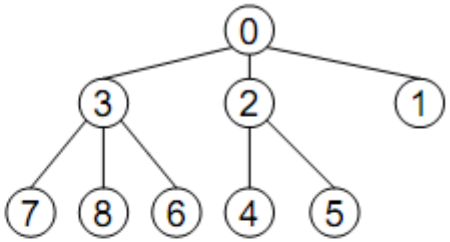




Graf Üzerinde Dolaşma

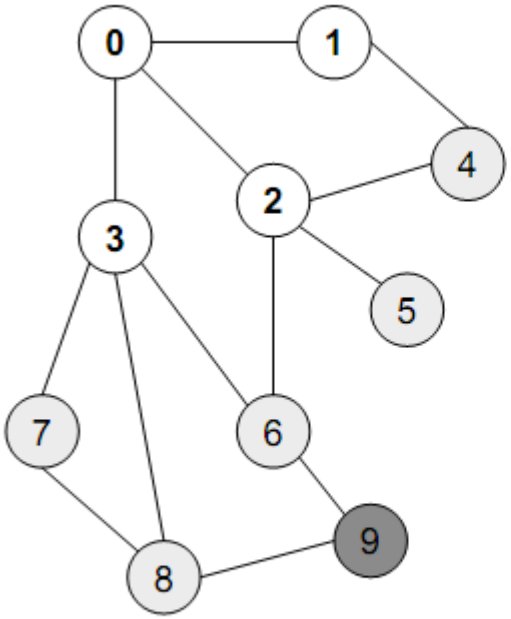
- **Breadth first arama**

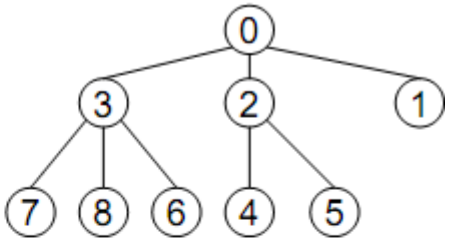


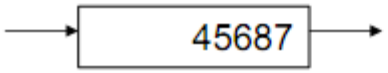


Graf Üzerinde Dolaşma

- **Breadth first arama**

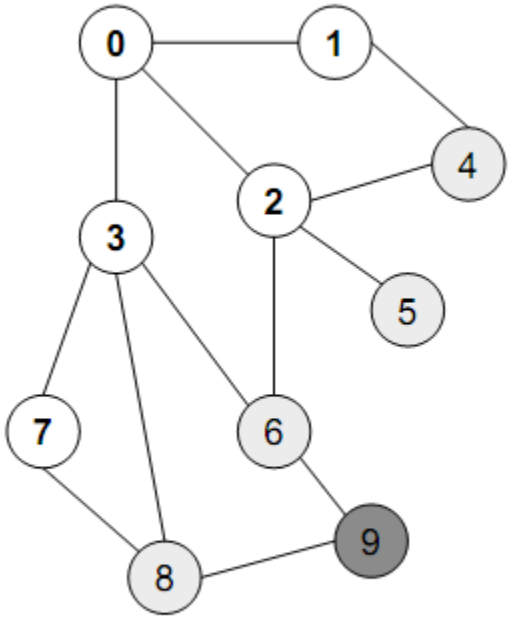


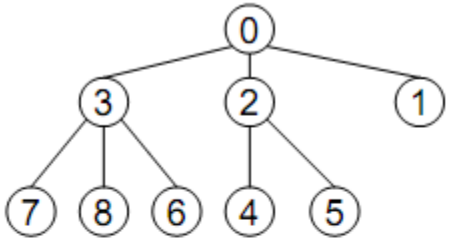


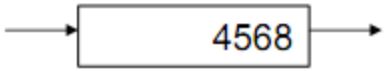


Graf Üzerinde Dolaşma

- **Breadth first arama**

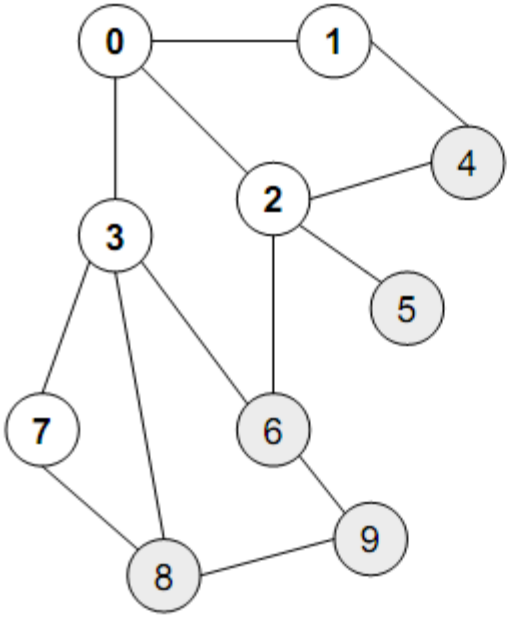


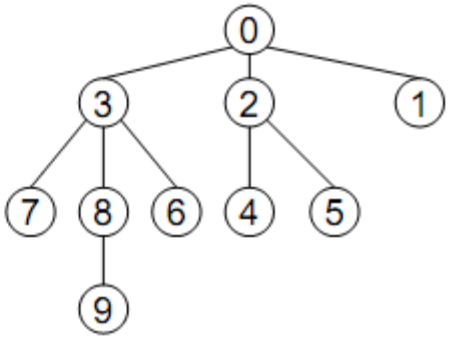


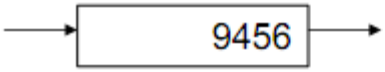


Graf Üzerinde Dolaşma

- **Breadth first arama**

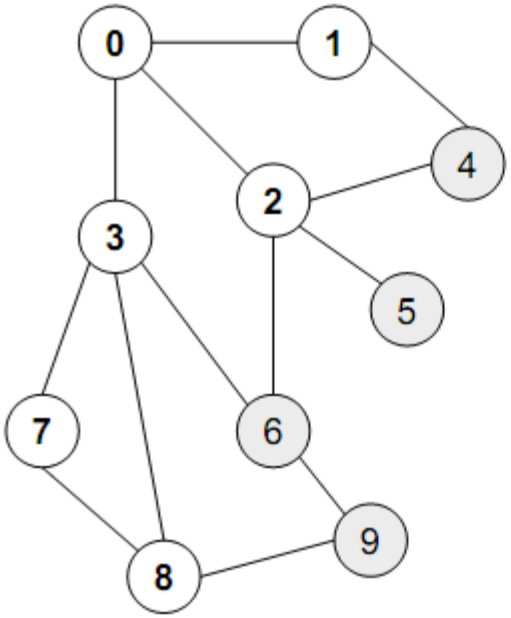


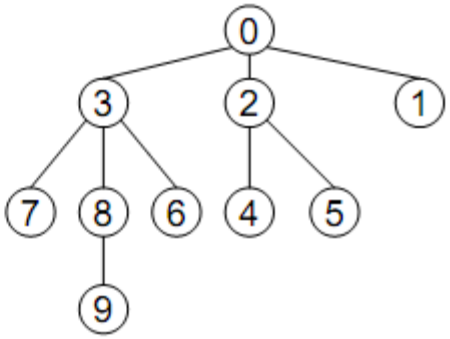


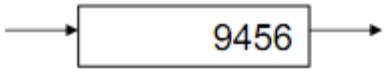


Graf Üzerinde Dolaşma

- **Breadth first arama**

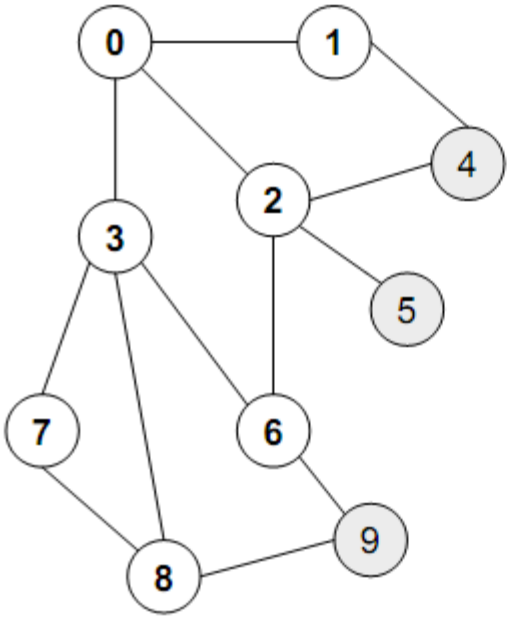


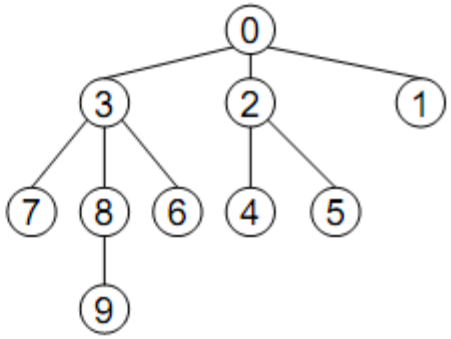




Graf Üzerinde Dolaşma

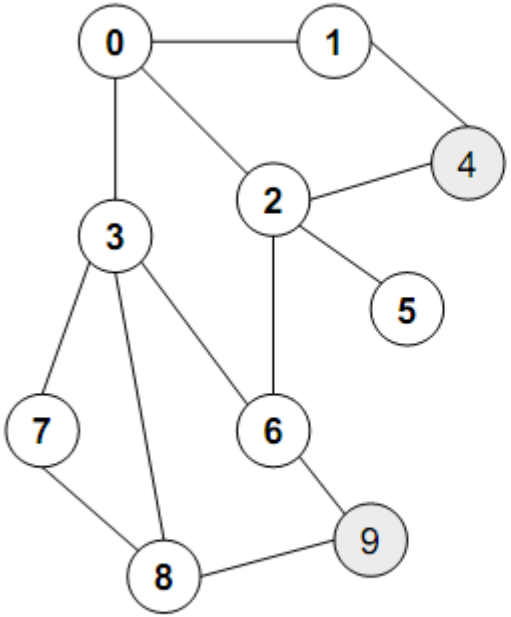
- **Breadth first arama**

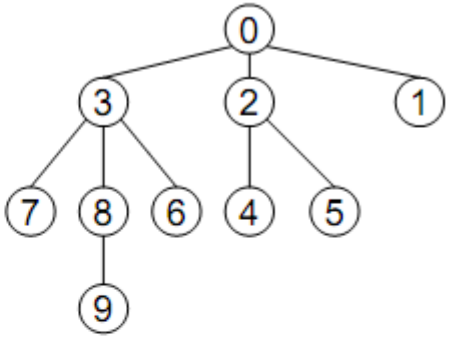




Graf Üzerinde Dolaşma

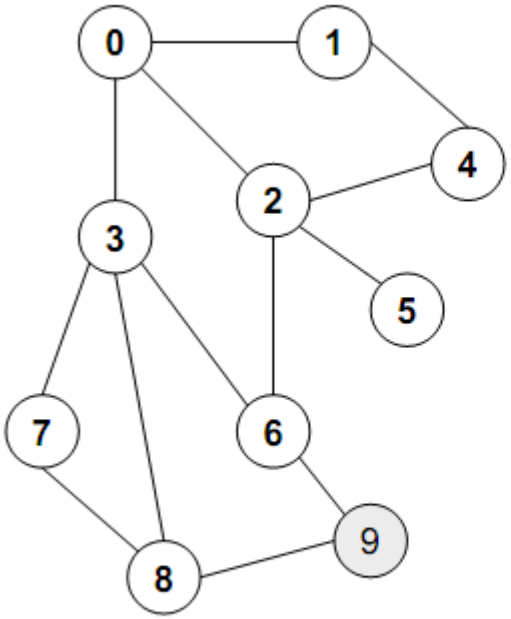
- **Breadth first arama**

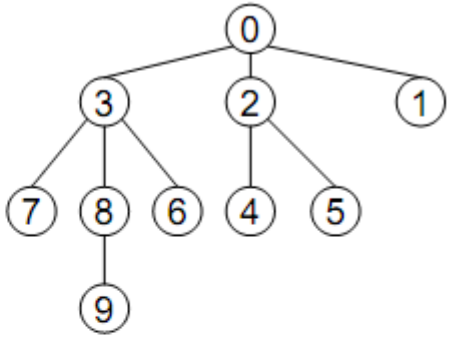




Graf Üzerinde Dolaşma

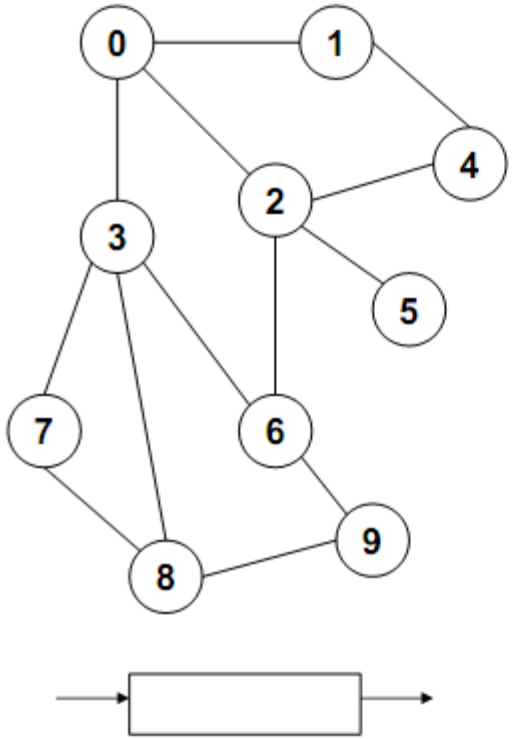
- **Breadth first arama**

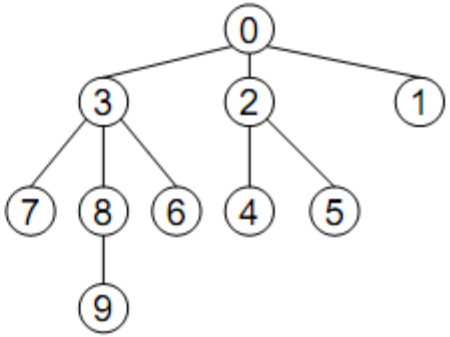




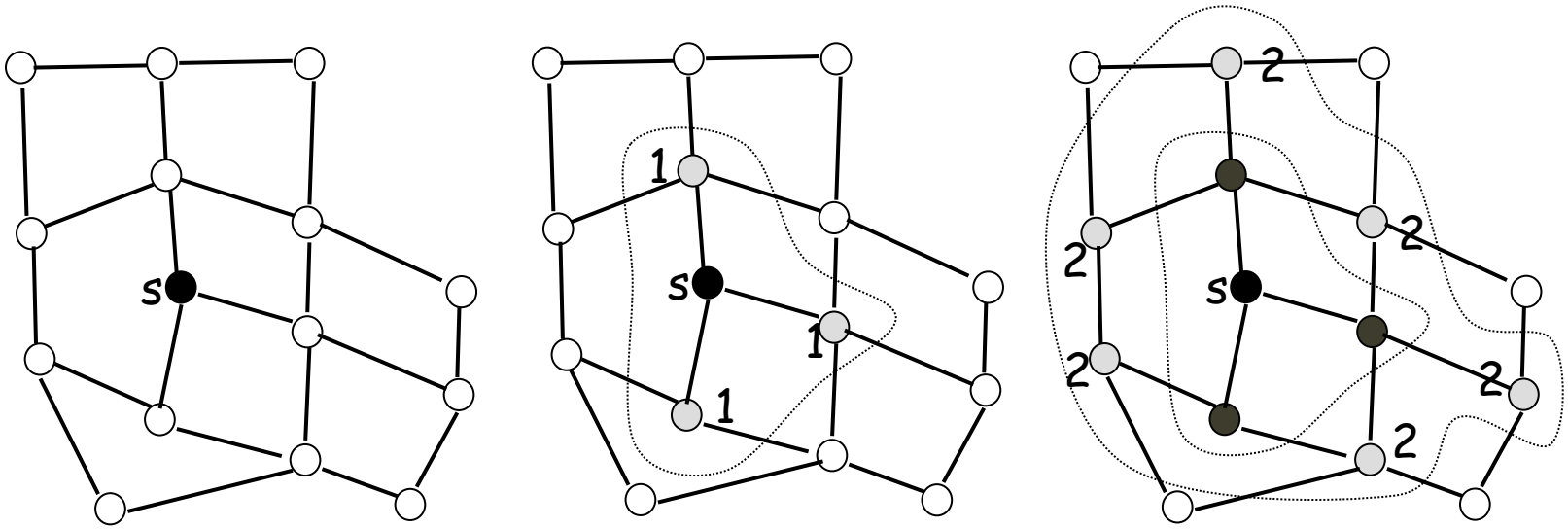
Graf Üzerinde Dolaşma

- **Breadth first arama**

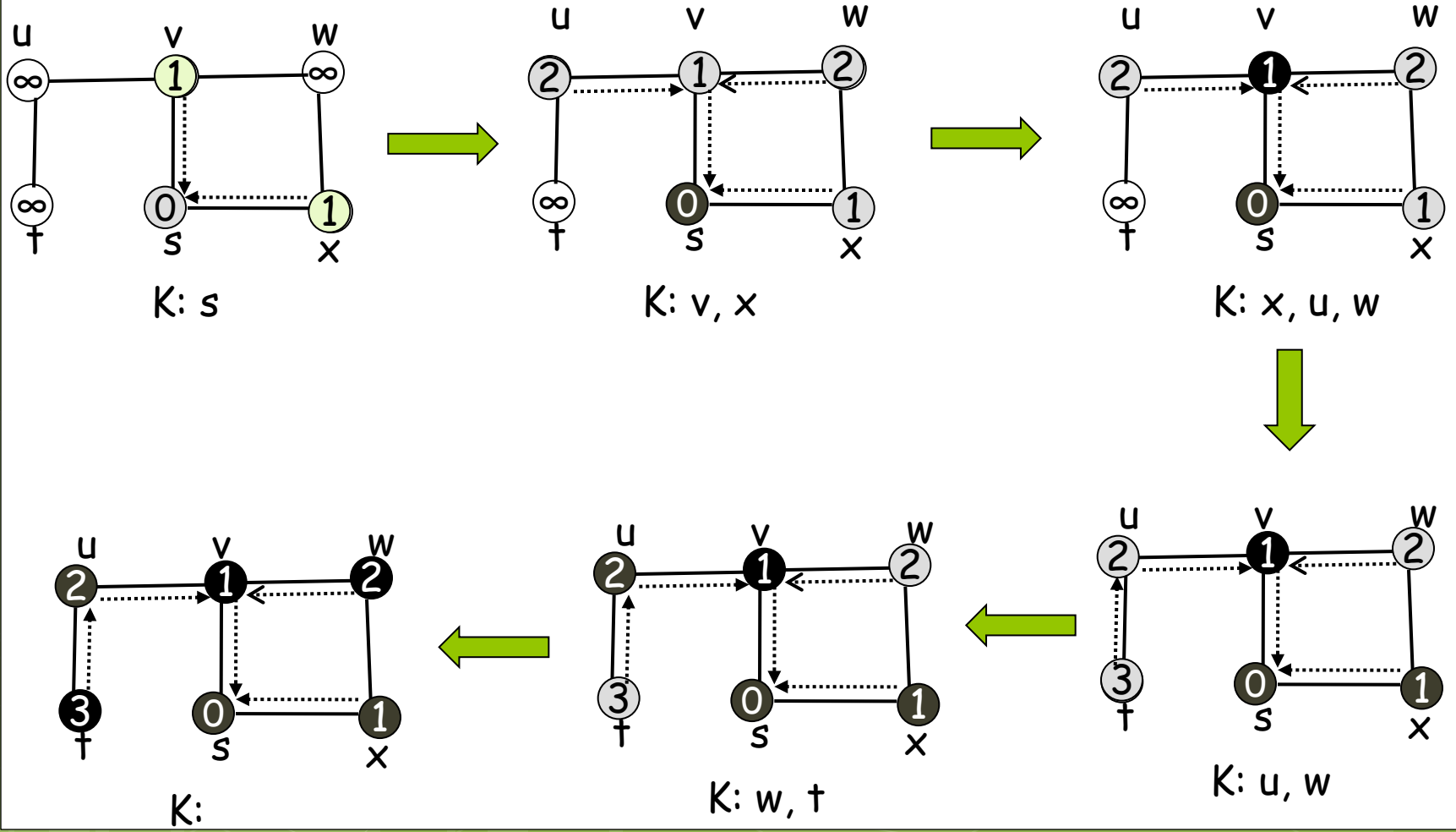




BFS-Örnek



BFS – Örnek


u
v
w
∞
1
∞
∞
0
1
t
s
x
K: s
2
1
2
∞
0
1
K: v, x
2
1
2
∞
0
1
K: x, u, w
2
1
2
3
0
1
K: u, w
2
1
2
3
0
1
K: w, t
2
1
2
3
0
1
K:

Graf Renklendirme

- Graf renklendirme, graf üzerinde birbirine komşu olan düğümlere farklı renk atama işlemidir; amaç, en az sayıda renk kullanılarak tüm düğümlere komşularından farklı birer renk vermektir. Renklendirmede kullanılan toplam renk sayısı kromatik (chromatik) sayı olarak adlandırılır.

- Uygulamada, graf renklendirmenin kullanılacağı alanların başında, ilk akla gelen, harita üzerindeki bölgelerin renklendirilmesi olmasına karşın, graf renklendirme bilgisayar biliminde ve günlük yaşamdaki birçok problemin çözümüne ciddi bir yaklaşımdır.

Graf Renklendirme

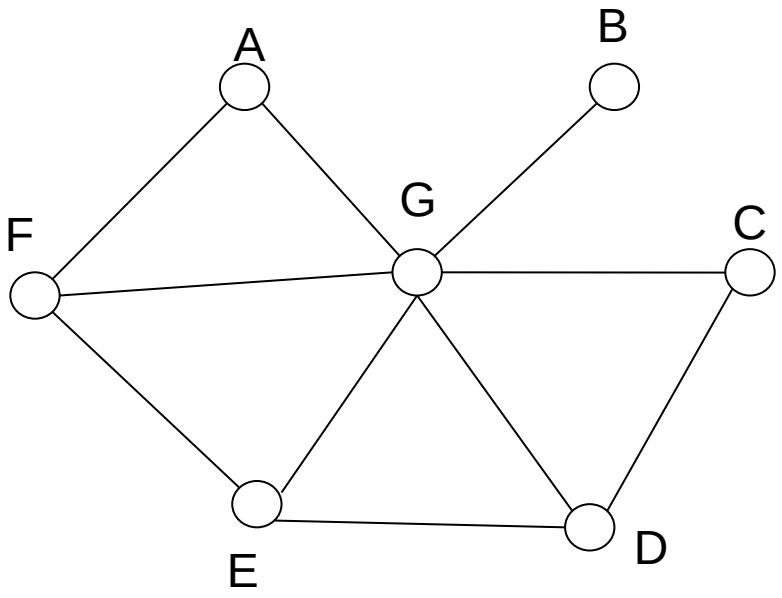
- Uygulama Alanları;
  - Harita renklendirme,
  - İşlemcilerin işlem sırasını belirleme,
  - Ders ve sınav programı ayarlama
  - Hava alanlarında iniş ve kalkış sırasını belirleme vs.

Graf Renklendirme

- Graf renklendirmede kullanılan algoritmaların başında Welch ve Powel algoritmasıdır.
- Welch ve Powel Algoritması:
  - Düğümler derecelerine göre büyükten küçüğe doğru sıralanır.
  - İlk renk birinci sıradaki düğüme atanır ve daha sonra aynı renk bitişik olamayacak şekilde diğer düğümlere verilir.
  - Bir sonraki renge geçilir ve aynı işlem düğümlerin tamamı renklendirilinceye kadar devam ettirilir.

Graf Renklendirme

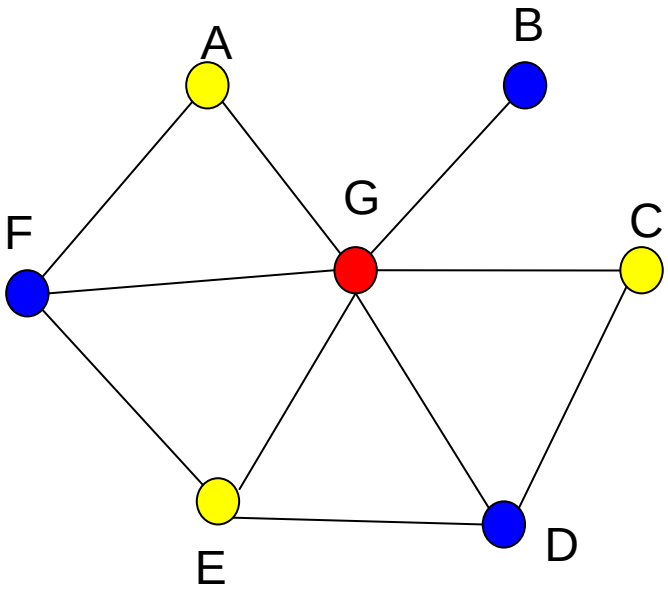
- Örnek: a) En az renk kullanılarak düğümleri renklendiriniz. Kromatik sayı kaç olur?



Düğüm	Derece
G	6
D	3
E	3
F	3
C	2
A	2
B	1

Graf Renklendirme

- Örnek: a) En az renk kullanılarak düğümleri renklendiriniz. Kromatik sayı kaç olur?

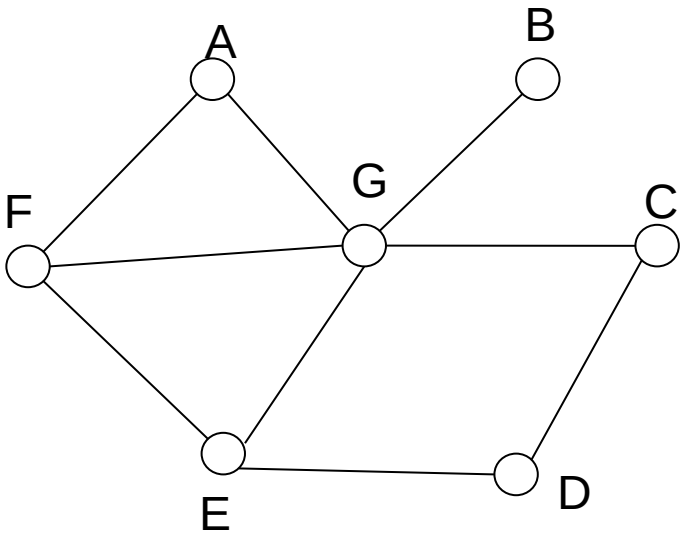


Düğüm	Derece
G	6
D	3
E	3
F	3
C	2
A	2
B	1

Kırmızı: G
Mavi: D,F,B
Sarı: E,A, C
Kromatik Sayı:3

Graf Renklendirme

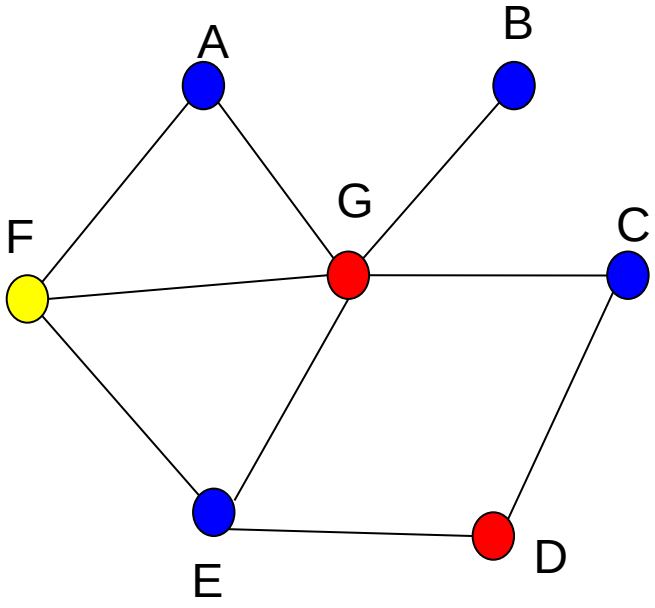
- b) G ile D arasındaki bağlantı kalkarsa yeni durum ne olur.



Düğüm	Derece
G	5
E	3
F	3
D	2
C	2
A	2
B	1

Graf Renklendirme

- b) G ile D arasındaki bağlantı kalkarsa yeni durum ne olur.



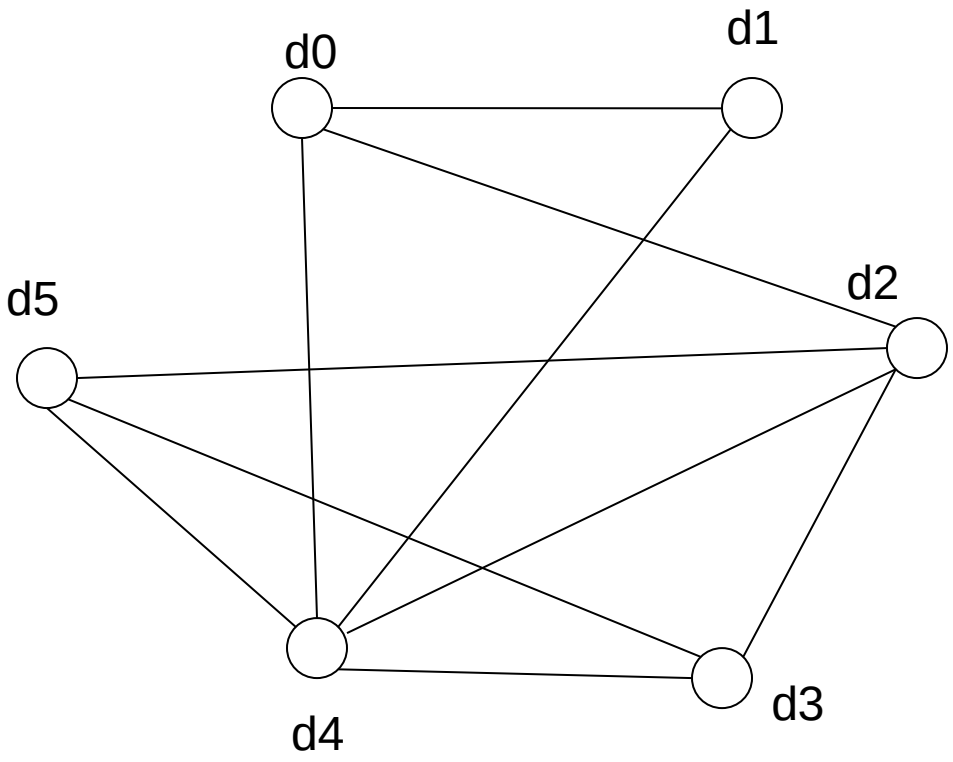
Düğüm	Derece
G	5
E	3
F	3
D	2
C	2
A	2
B	1

Kırmızı: G,D
Mavi:E,C,B,A
Sarı: F

Graf Renklendirme

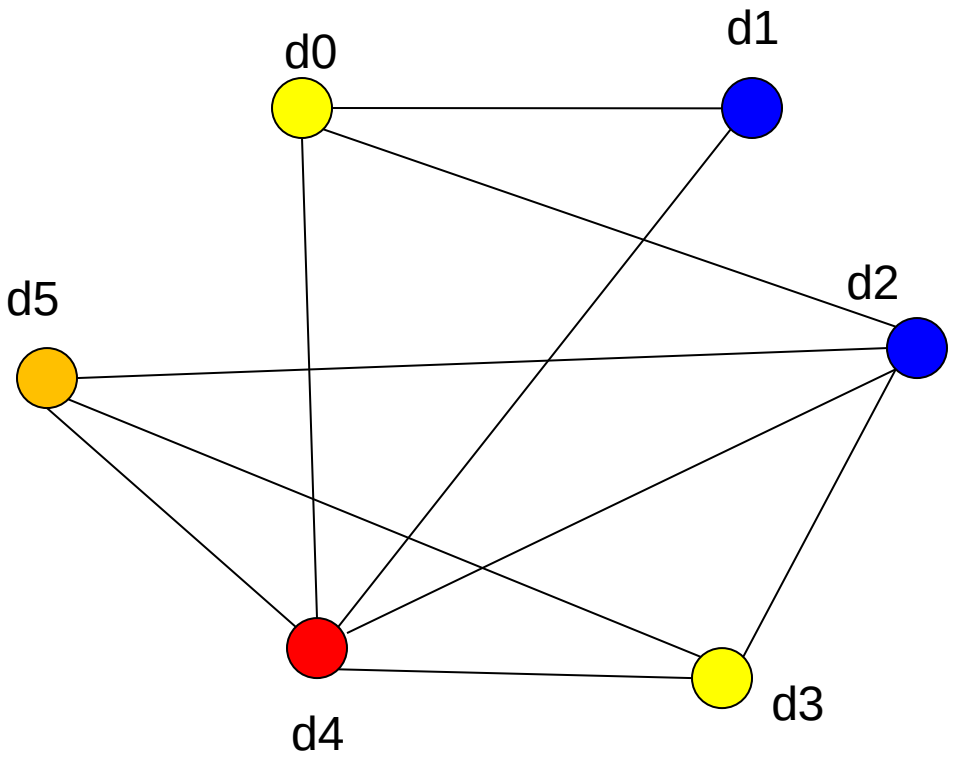
- Örnek: Farklı sınıflardaki öğrencilerin sınavlarını hazırlarken öğrencilerin gireceği sınavların çakışmasını engellemek için en az kaç farklı oturum yapılmalıdır
  - Öğrenci1: d0,d1,d4
  - Öğrenci2: d0,d2,d4
  - Öğrenci3: d2,d3,d5
  - Öğrenci4: d3,d4,d5

Graf Renklendirme



Düğüm	Derece
d4	5
d2	4
d0	3
d3	3
d5	3
d1	2

Graf Renklendirme



Düğüm	Derece
d4	5
d2	4
d0	3
d3	3
d5	3
d1	2

Kırmızı: d4
Mavi: d1,d2
Sarı: d0,d3
Turuncu: d5

En Küçük Yol Ağacı (Minimum Spanning Tree)

En Kısa Yol Problemi

- G = (D, K) grafı verilsin ve s başlangıç düğümünden V düğümüne giden en az maliyetli yol bulma.

- Farklı varyasyonlar mevcut
  - Ağırlıklı ve ağırlıksız graflar
  - Sadece pozitif ağırlık veya negatif ağırlığın da olması.

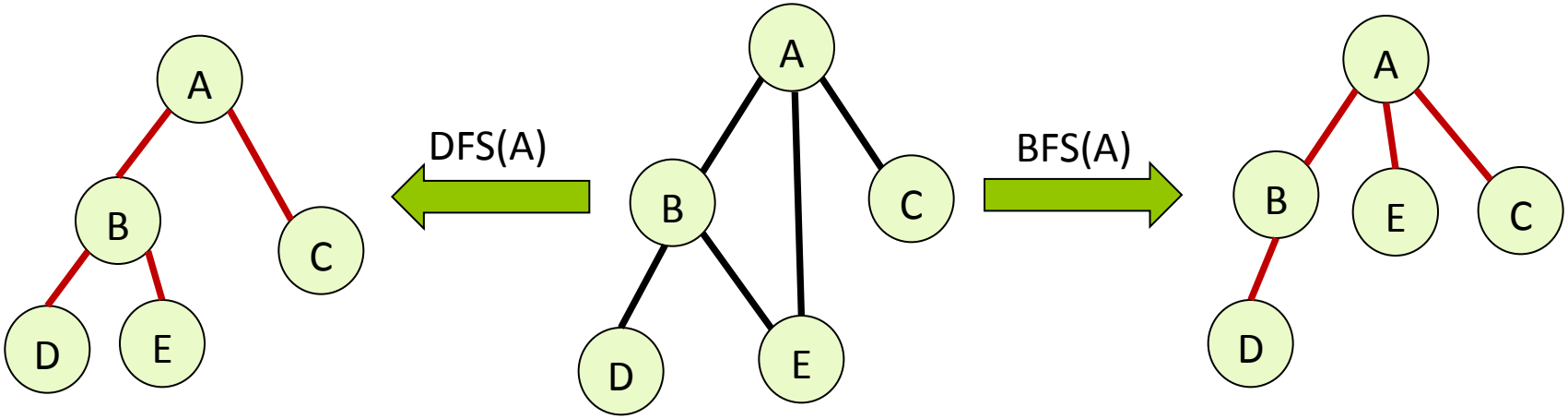
En Küçük Yol Ağacı (Minimum Spanning Tree)

- Yol ağacı, bir graf üzerinde tüm düğümleri kapsayan ağaç şeklinde bir yoldur.
  - Ağaç özelliği olduğu için kapalı çevrim(çember) içermez.

- Bir graf üzerinde birden çok yol ağacı olabilir. En az maliyetli olan en küçük yol ağacı (minimum spanning tree) olarak adlandırılır.

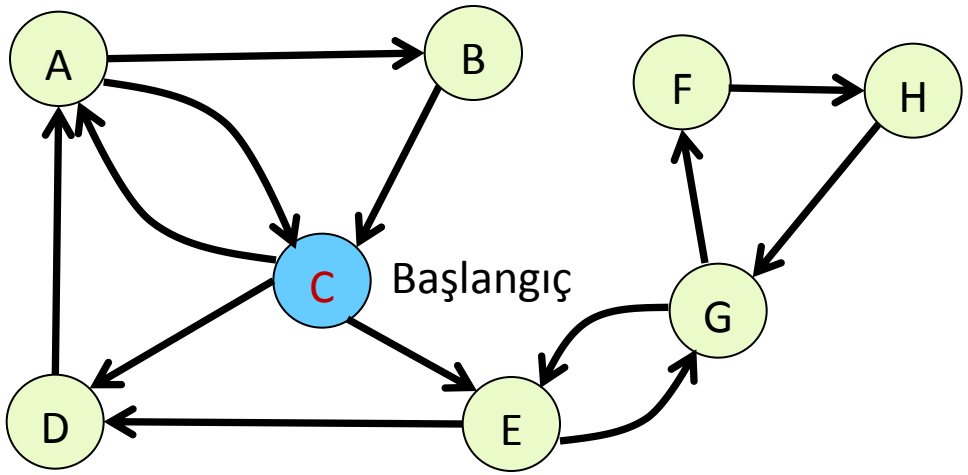
MST Hesaplama – Ağırlıksız Graf

- Graf ağırlıksızsa veya tüm kenarların ağırlıkları eşit ise MST nasıl bulunur?
  - BSF veya DSF çalıştırın oluşan ağaç MST'dir



Ağırlıksız En Kısa Yol Problemi

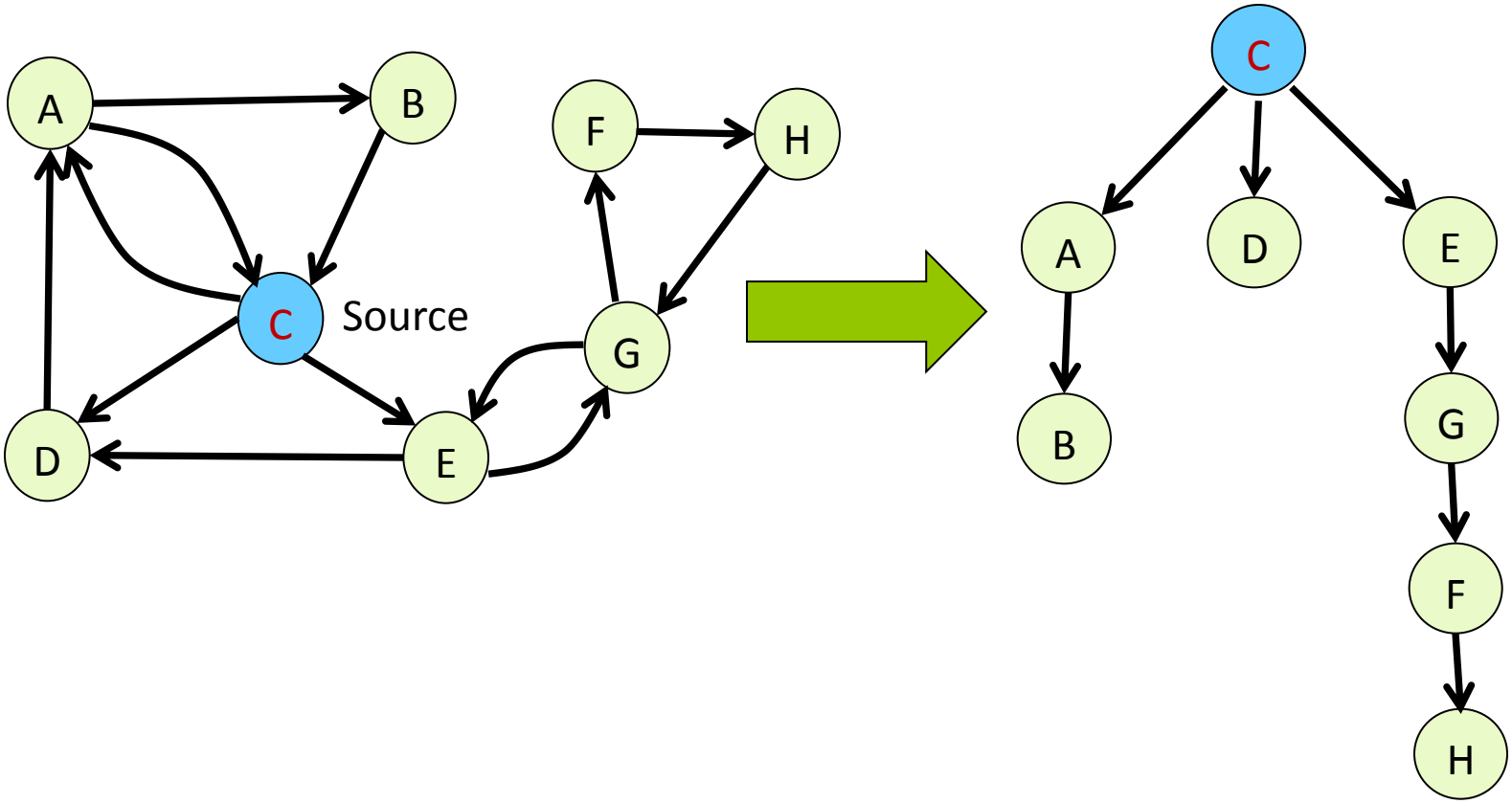
- Problem: G = (D, K) ağırlıksız grafında s başlangıç düğümü veriliyor ve s'den diğer düğümlere giden en kısa yol nasıl bulunur.



- C'den diğer düğümlere giden en kısa yolu bulun?

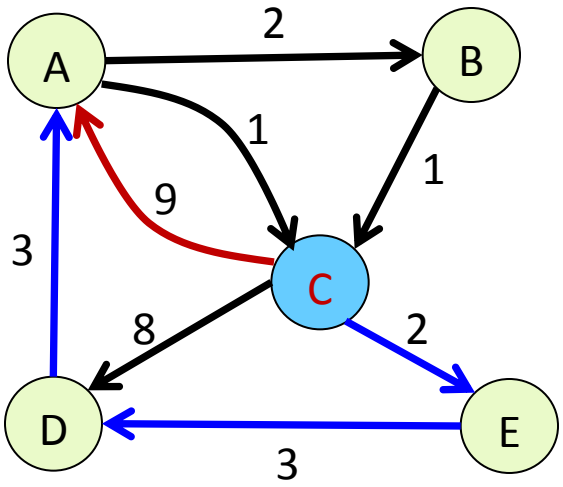
BFS Tabanlı Çözüm

- S'den başla BFS algoritmasını uygula



Ağırlıklı Graflarda En Kısa Yol Problemi

- BFS algoritması bu graf içinde çalışır mı?
  - Hayır!



- C den A'ya en kısa yol:
  - C->A (uzunluk: 1, maliyet: 9)
  - BFS ile hesaplandı
- C den A ' ya en az maliyetli yol:
  - C->E->D->A (uzunluk: 3, maliyet: 8)
  - Peki nasıl hesaplayacağız?

Algoritmalar

- En küçük yol ağacını belirlemek için birçok algoritma geliştirilmiştir.
  - **Kruskal'ın Algoritması:** Daha az maliyetli kenarları tek tek değerlendirerek yol ağacını bulmaya çalışır. Ara işlemler birden çok ağaç oluşturabilir.
  - **Prim'in Algoritması:** En az maliyetli kenardan başlayıp onun uçlarından en az maliyetle genişleyecek kenarın seçilmesine dayanır. Bir tane ağaç oluşur.
  - **Sollin'in Algoritması:** Doğrudan paralel programlamaya yatkındır. Aynı anda birden çok ağaçla başlanır ve ilerleyen adımlarda ağaçlar birleşerek tek bir yol ağacına dönüşür.
  - **Dijkstra Algoritması:**
    - Ağırlıklı ve yönlü graflar için geliştirilmiştir.
    - Graf üzerindeki kenarların ağırlıkları 0 veya sıfırdan büyük sayılar olmalıdır.
    - Negatif ağırlıklar için çalışmaz.
  - **Bellman ve Ford Algoritması:**
    - Negatif ağırlıklı graflar için geliştirilmiştir.
  - **Floyd Algoritması**

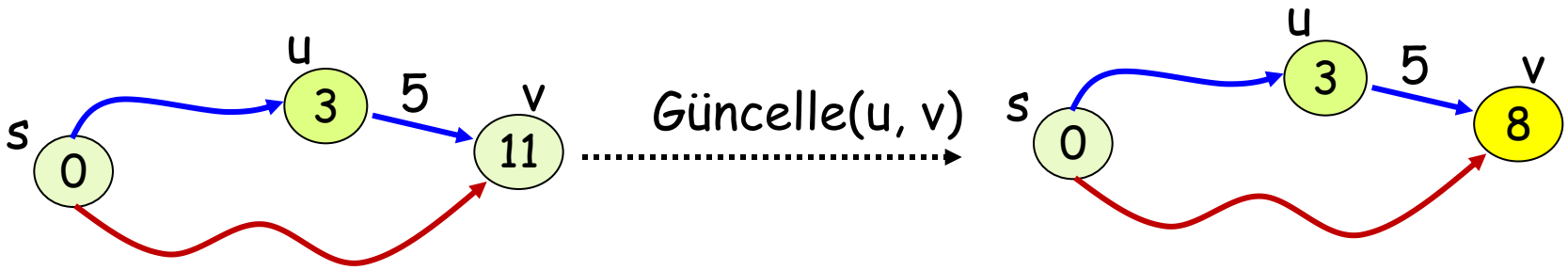
Dijkstra Algoritması

- Başlangıç olarak sadece başlangıç düğümünün en kısa yolu bilinir. (0 dır.)

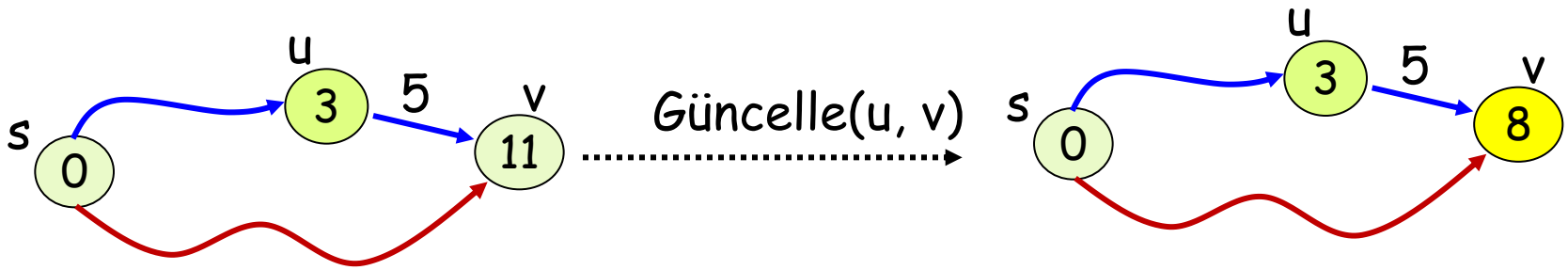
- Tüm düğümlerin maliyeti bilinene kadar devam et.
  1. O anki bilinen düğümler içerisinden en iyi düğümü şeç. (en az maliyetli düğümü seç, daha sonra bu düğümü bilinen düğümler kümesine ekle)
  2. Seçilen düğümün komşularının maliyetlerini güncelle.

Güncelleme

- Adım-1 de seçilen düğüm u olsun.
- u düğümünün komşularının maliyetini güncelleme işlemi aşağıdaki şekilde yapılır.
  - s'den v'ye gitmek için iki yol vardır.
  - Kırmızı yol izlenebilir. Maliyet 11.
  - veya mavi yol izlenebilir. Önce s'den u'ya 3 maliyeti ile gidilir. Daha sonra (u, v) kenarı üzerinden 8 maliyetle v'ye ulaşılır.

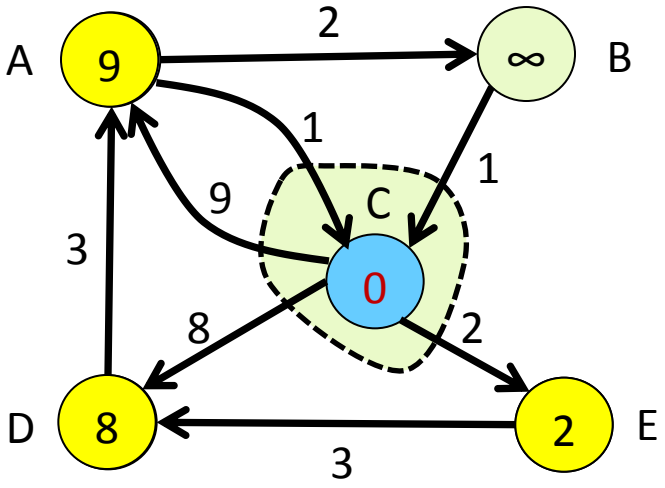


Güncelleme - Kaba Kod




```
Guncelle(u, v){
        if (maliyet[u] + w(u, v) < maliyet[v]){        // U üzerinden yol daha kısa ise
              maliyet[v] = maliyet[u] + w(u, v);        // Evet! Güncelle
              pred[v] = u;                              // u'dan geldiğimizi kaydet.
        }
}
```

Dijkstra'nın Algoritması



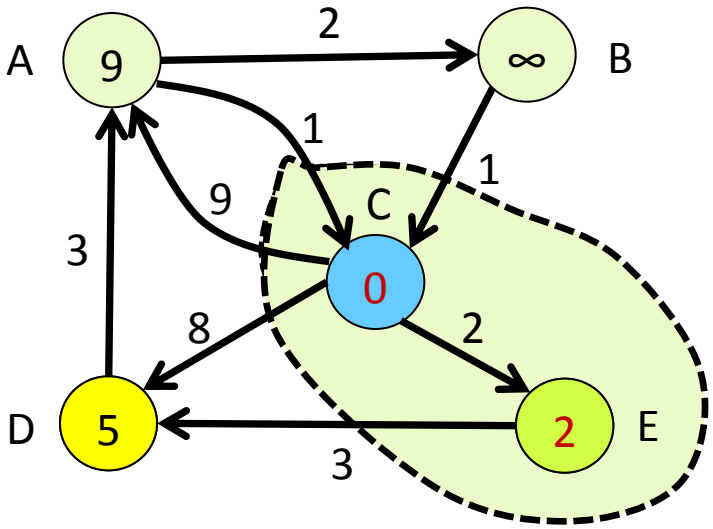
1. O anki en iyi düğümü seç – C
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu A: 0 + 9 < ∞ → maliyet(A) = 9

Komşu D: 0 + 8 < ∞ → maliyet(D) = 8

Komşu E: 0 + 2 < ∞ → maliyet(E) = 2

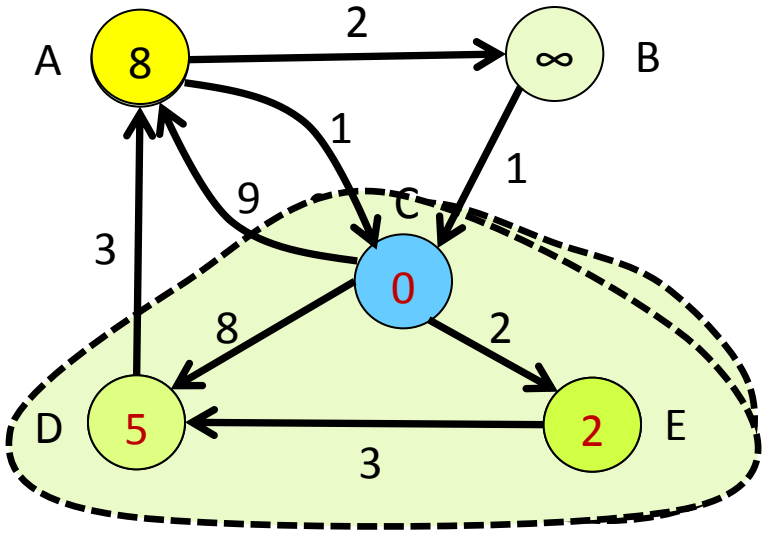
Dijkstra'nın Algoritması



1. O anki en iyi düğümü seç – E
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu D: 2 + 3 = 5 < 8 → maliyet(D) = 5

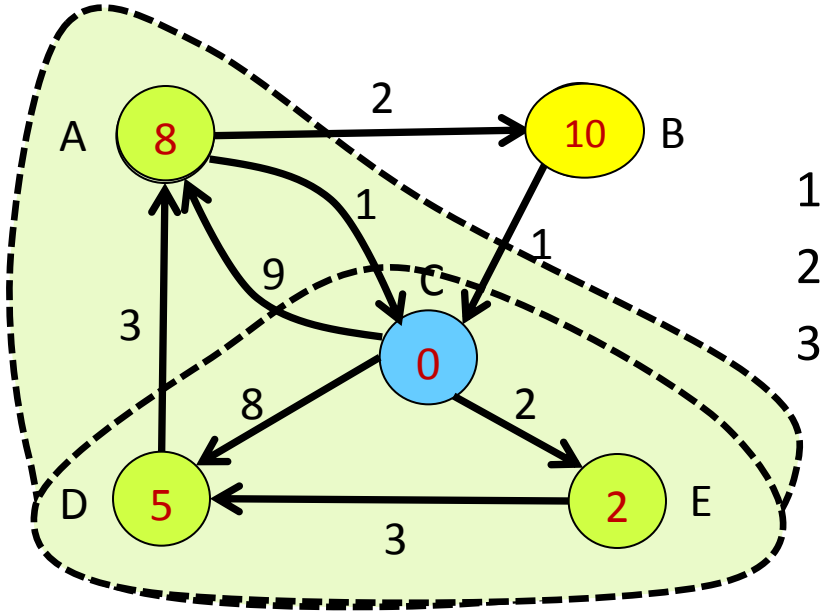
Dijkstra'nın Algoritması



1. O anki en iyi düğümü seç – D
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu A: 5 + 3 = 8 < 9 → maliyet(A) = 8

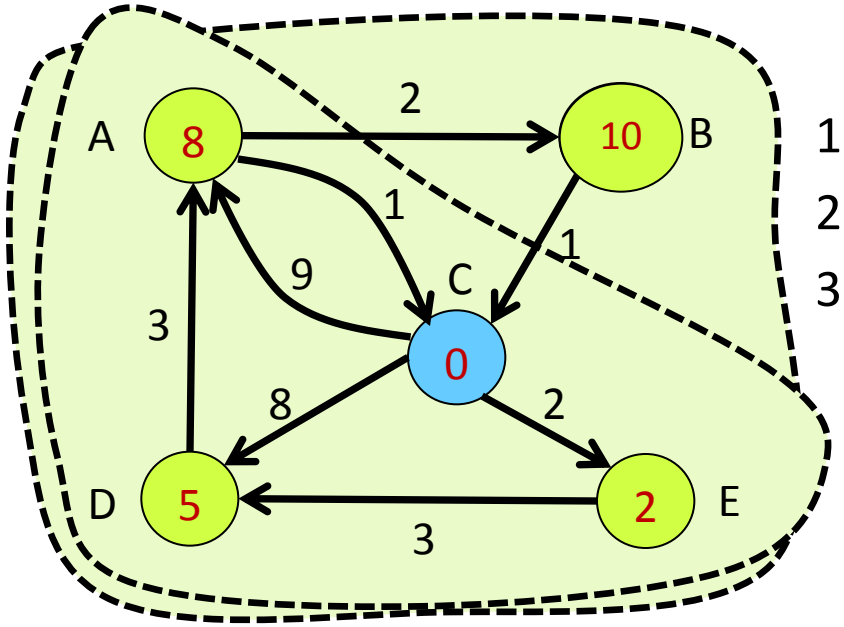
Dijkstra'nın Algoritması



1. O anki en iyi düğümü seç – A
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu B: 8 + 2 = 10 < ∞ → maliyet(B) = 10

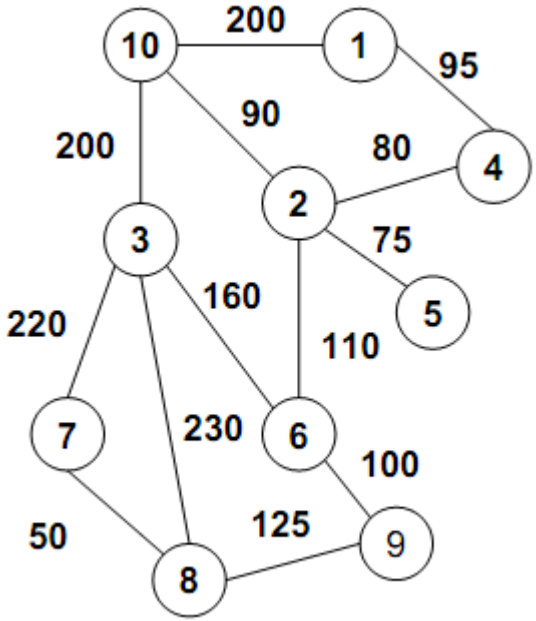
Dijkstra'nın Algoritması



1. O anki en iyi düğümü seç – B
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

ÖDEV

- Programları Java/C # programları ile gerçekleştiriniz.
- 1- 10 tane şehir için bir graf yapısı oluşturunuz. Her şehirden komşu şehirlere olan uzaklık kenar ağırlıkları olarak kullanılacaktır. Herhangi bir şehirden başlayarak tüm şehirleri dolaşmak için gerekli olan algoritmayı ve grafiksel çözümü BFS ve DFS ile yapınız.
- 2-Dijkstra algoritmasını kullanarak en kısa yolu bulmak için gerekli olan algoritmayı ve grafiksel çözümü yapınız.
- 3- Bölümünüze ait haftalık ders ve sınav programının çakışmadan yerleşebilmesi için gerekli programı yazınız.
- 4-Herhangi bir labirent için gerekli komşulukları belirleyip, istenilen konumdan çıkışa gitmesi için gerekli yolu bulan programı yazınız.
- 5- Bölümünüze ait sınav programını çakışmadan yerleştirmek için gerekli olan çizimi gerçekleştirip kromatik sayıyı bulunuz. Ayrıca gerekli program kodlarını yazınız.



Örnekler

Örnekler

- **Depth-First Search**
- **class Node**
- **{ int label; /* vertex label*/ Node next; /* next node in list*/**
- **Node(int b) { label = b; } // constructor**
- **}**
- **class Graph**
- **{ int size; Node adjList[]; int mark[];**
- **Graph(int n) // constructor**
- **{ size = n; adjList = new Node[size];**
- **mark = new int[size]; // elements of mark are initialized to 0**
- **}**
- **public void createAdjList(int a[][]) // create adjacent lists**
- **{ Node p; int i, k;**
- **for(i = 0; i < size; i++)**
- **{ p = adjList[i] = new Node(i); //create first node of ith adj. list**
- **for(k = 0; k < size; k++)**
- **{ if(a[i][k] == 1)**
- **{ p.next = new Node(k); /* create next node of ith adj. List*/ p = p.next; }**
- **}**
- **}**
- **}**

Örnekler

```
public void dfs(int head) // recursive depth-first search
{ Node w; int v;    mark[head] = 1; System.out.print( head + " ");
   w = adjList[head];
  while( w != null)
  {  v = w.label;
   if( mark[v] == 0 ) dfs(v);
    w = w.next;
  }
 }
}
```

```
Dfs.java
class Dfs
{   public static void main(String[] args)
   { Graph g = new Graph(5); // graph is created with 5 nodes
      int a[][] = { {0,1,0,1,1}, {1,0,1,1,0}, {0,1,0,1,1},   {1,1,1,0,0}, {1,0,1,0,0}};
      g.createAdjList(a);
     g.dfs(0); // starting node to dfs is0 (i.e., A)
   }
}
```

Output of this program is: 0 1 2 3 4

Here, 0 is for A, 1 is for B, 2 is for C, 3 is for D, and 4 is for E

Örnekler

- **Breadth-First Search**
- class Node
- { int label; /* vertex label*/ Node next; /* next node in list*/
- Node(int b) { label = b; } // constructor
- }
- class Graph
- { int size; Node adjList[]; int mark[];
- Graph(int n) // constructor
- { size = n; adjList = new Node[size]; mark = new int[size]; }
- public void createAdjList(int a[][]) // create adjacent lists
- { Node p; int i, k;
- for(i = 0; i < size; i++)
- { p = adjList[i] = new Node(i);
- for(k = 0; k < size; k++)
- { if(a[i][k] == 1) { p.next = new Node(k); p = p.next; }
- } // end of inner for-loop
- } // end of outer for-loop
- } // end of createAdjList()

Örnekler

```
public void bfs(int head)
{ int v;  Node adj;     Queue q = new Queue(size);
v = head;  mark[v] = 1;  System.out.print(v + ” “);    q.qinsert(v);
while( !q.IsEmpty() ) // while(queue not empty)
{  v = q.qdelete();       adj = adjList[v];
   while( adj != null )
 { v = adj.label;
   if( mark[v] == 0 )   { q.qinsert(v); mark[v] = 1; System.out.print(v + ” “);  }
   adj = adj.next;   }
}   }    } // end of Graph class

class Queue
{ private int maxSize; // max queue size
private int[] que; // que is an array of integers
private int front;  private int rear;   private int count; // count of items in queue
public Queue(int s) // constructor
{ maxSize = s;  que = new int[maxSize];   front = rear = -1;  }
```

Örnekler

- public void qinsert(int item)
- { if(rear == maxSize-1) System.out.println("Queue is Full");
- else { rear = rear + 1; que[rear] = item; if(front == -1) front = 0; }
- }
- public int qdelete()
- { int item;
- if(IsEmpty()) { System.out.println("\n Queue is Empty"); return(-1); }
- item = que[front];
- if(front == rear) front = rear = -1;
- else front = front+1; return(item); }
- public boolean IsEmpty() { return(front == -1); } }
- class BfsDemo
- { public static void main(String[] args)
- { Graph g = new Graph(5);
- int a[][] = { {0,1,0,1,1}, {1,0,1,1,0}, {0,1,0,1,1}, {1,1,1,0,0}, {1,0,1,0,0}};
- g.createAdjList(a); g.bfs(0); }
- }
- Output of this program is: 0 1 3 4 2

Örnekler

- **Depth First Arama**
- #include <stdio.h>
- #define max 10
- void buildadjm(int adj[][max], int n)
- {
- int i,j;
- for(i=0;i<n;i++)
- for(j=0;j<n;j++) {
- printf("%d ile %d komşu ise 1 değilse 0 gir \n", i,j);
  scanf("%d",&adj[i][j]); }
- }

Örnekler

- void dfs(int x,int visited[],int adj[][max],int n)
- {
- int j;
- visited[x] = 1;
- printf("Ziyaret edilen düğüm numarası %d\n",x);
- for(j=0;j<n;j++)
- if(adj[x][j] ==1 && visited[j] ==0)
  dfs(j,visited,adj,n);
- }

Örnekler

- void main() {
- int adj[max][max],node,n,i, visited[max];
- printf(“Düğüm sayısını girin (max = %d)\n",max);
- scanf("%d",&n);
- buildadjm(adj,n);
- for(i=0; i<n; i++)
- visited[i] =0;
- for(i=0; i<n; i++) if(visited[i] ==0) dfs(i,visited,adj,n); }

Örnekler

- Breadth-first
- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 10
- struct node
- {
- int data;
- struct node *link;
- };

Örnekler

- **Breadth-first**
- void buildadjm(int adj[][MAX], int n)
- {
- int i,j;
- printf("enter adjacency matrix \n",i,j);
  for(i=0;i<n;i++)
- for(j=0;j<n;j++) scanf("%d",&adj[i][j]);
- }

Örnekler

- struct node *addqueue(struct node *p,int val)
- {
- struct node *temp;
- if(p == NULL) {
- p = (struct node *) malloc(sizeof(struct node));
- if(p == NULL) {
- printf("Cannot allocate\n"); exit(0);
  }
- p->data = val;
- p->link=NULL; }

Örnekler

- else { temp= p;
- while(temp->link != NULL){
- temp = temp->link;}
- temp->link = (struct node*)malloc(sizeof(struct node));
- temp = temp->link;
- if(temp == NULL) {
- printf("Cannot allocate\n"); exit(0); }
- temp->data = val; temp->link = NULL;}
- return(p);
- }

Örnekler

- struct node *deleteq(struct node *p,int *val){
- struct node *temp;
- if(p == NULL) {
- printf("queue is empty\n"); return(NULL); }
- *val = p->data;
- temp = p;
- p = p->link;
- free(temp);
- return(p); }

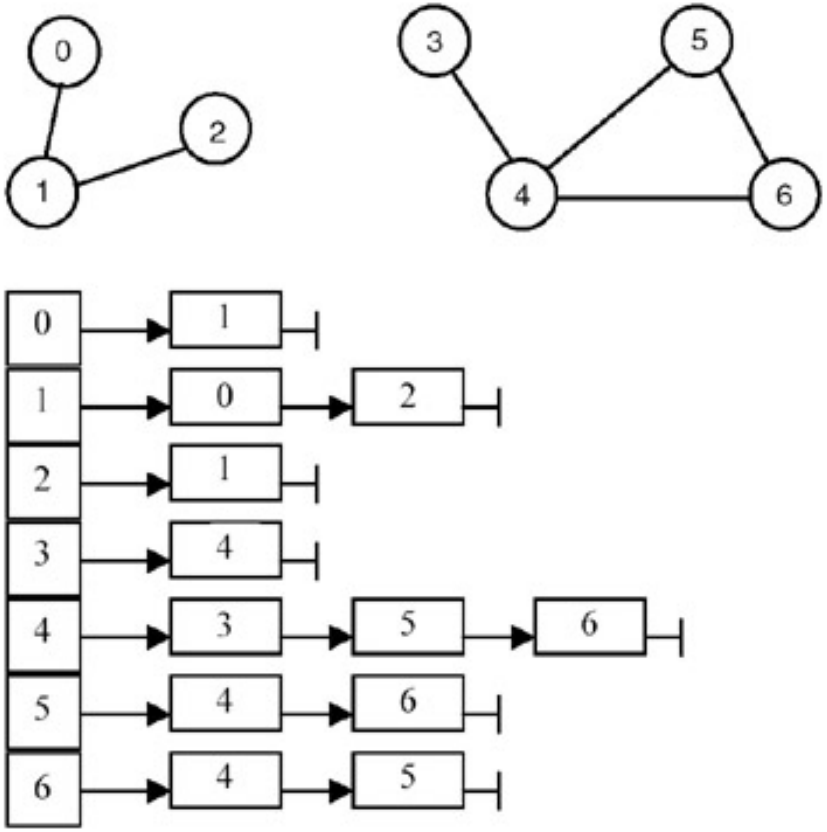
Örnekler

```
void bfs(int adj[][MAX], int x,int visited[], int n, struct node **p)
{

int y,j,k;    *p = addqueue(*p,x);
do {
*p = deleteq(*p,&y);

if(visited[y] == 0)  {
printf("\nnode visited = %d\t",y);
visited[y] = 1;
for(j=0;j<n;j++)
if((adj[y][j] ==1) && (visited[j] == 0))
*p = addqueue(*p,j);
      }

}while((*p) != NULL);}
```

Örnekler

- void main() {
- int adj[MAX][MAX];
- int n;
- struct node *start=NULL;
- int i, visited[MAX];
- printf("enter the number of nodes in graph maximum = %d\n",MAX);
- scanf("%d",&n);
- buildadjm(adj,n);
- for(i=0; i<n; i++) visited[i] =0;
- for(i=0; i<n; i++) if(visited[i] ==0) bfs(adj,i,visited,n,&start);
- }

Örnekler



Örnekler

- #include <stdio.h>
- #include <stdlib.h>
- #define MAXVERTICES 20
- #define MAXEDGES 20
- typedef enum {FALSE, TRUE, TRISTATE} bool;
- typedef struct node node;
- struct node {
- int dst;
- node *next;};

Örnekler

```
void printGraph( node *graph[], int nvert ) {
  /*    * graph çiz. */
  int i, j;
  for( i=0; i<nvert; ++i ) {
     node *ptr;
     for( ptr=graph[i]; ptr; ptr=ptr->next )
        printf( "[%d] ", ptr->dst );
     printf( "\n" );
  }
}
```

Örnekler

```
void insertEdge( node **ptr, int dst ) {
    /*
     * Başlamak için yeni düğüm ekle.
     */
    node *newnode = (node *)malloc(sizeof(node));
    newnode->dst = dst;
    newnode->next = *ptr;
    *ptr = newnode;
}
```

Örnekler

- void buildGraph (node *graph[], int edges[2][MAXEDGES], int nedges) {
- /* Graph kenar dizilerinde bitişik olanların listesi ile doldur. */
- int i;
- for(i=0; i<nedges; ++i) {
- insertEdge(graph+edges[0][i], edges[1][i]);
- insertEdge(graph+edges[1][i], edges[0][i]); // yönsüz graph.
- }
- }

Örnekler

```
void dfs( int v, int *visited, node *graph[] ) {
    /*
     * Tekrarlamalı olarak v kullanılarak ziyaret edilen graph
     * TRISTATE olarak işaretlenmekte
     */
    node *ptr;
    visited[v] = TRISTATE;
    //printf( "%d \n", v );
    for( ptr=graph[v]; ptr; ptr=ptr->next )
      if( visited[ ptr->dst ] == FALSE )
          dfs( ptr->dst, visited, graph );
}
```

Örnekler

```
void printSetTristate( int *visited, int nvert ) {
    /*
     * Tüm düğümler ziyaret edilmiş ise (TRISTATE) TRUE değerini ata.
     */
    int i;
    for( i=0; i<nvert; ++i )
      if( visited[i] == TRISTATE ) {
          printf( "%d ", i );
          visited[i] = TRUE;
      }
    printf( "\n\n" );
}
```

Örnekler

```
void compINC(node *graph[], int nvert) {
   /* Birbirine bağlı tüm bileşenlierin sunulduğu graph listesi.   */
   int *visited;
   int i;
   visited = (int *)malloc( nvert*sizeof(int) );
   for( i=0; i<nvert; ++i )
      visited[i] = FALSE;
   for( i=0; i<nvert; ++i )
      if( visited[i] == FALSE ) {
         dfs( i, visited, graph );
         // print all vertices which are TRISTATE. and mark them to TRUE.
         printSetTristate( visited, nvert );
      }
  free( visited );
}
```

Örnekler

```
int main() {
    int edges[][MAXEDGES] = { {0,2,4,5,5,4},
                              {1,1,3,4,6,6}
                            };
    int nvert = 7; // hiç bir düğüm.
    int nedges = 6; // ve hiçbir kenarın olmadığı graph.
    node **graph = (node **)calloc( nvert, sizeof(node *) );
    buildGraph( graph, edges, nedges );
    printGraph( graph, nvert );
    compINC( graph, nvert );
    return 0;
}
```

Örnekler