YBS Ansiklopedi
www.YBSAnsiklopedi.com
Cilt 2, Sayı 2, Haziran 2015

Çizge Teorisi (Graph Theory)

Sadi Evren SEKER

Istanbul Medeniyet University, Department of Business, Management Information Systems Program

ÖZET

Çizge Teorisi çok farklı disiplinlerin çalışma alanına girmektedir. Sosyolojiden, bilgisayar bilimlerine, işletmeden, endüstri mühendisliğine kadar çok geniş alanlarda kullanımı olan teori, basitçe bir gerçek hayat probleminin çizge ile modellenmesini amaçlamaktadır. Model oluşturulduktan sonra çizge teorisinde bulunan yöntemler kullanılarak problem çözülebilmekte ve ardından da tekrar gerçek hayata uygulanabilmektedir. Bu yazı kapsamında, çizge teorisinin genel kavramlarına giriş yapılarak terminoloji hakkında bilgi verilecek ve bazı problemlerin nasıl çözüldüğü gösterilecektir. Yazının başlangıç kısmında temel terminolojiye yer verilmişken ilerleyen kısımlarında Öyler yolu, Hamilton yolu, Prüfer dizilimi, Ağaçlar ve özel olarak ikili ve ikili arama ağaçları, yığıtlar, asgari tarama ağacı ve çözümü için Kruskal ve Prims algoritmalarına yer verilmiştir.

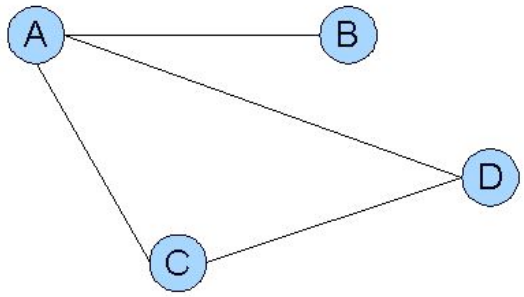
Çizge teorisi (graph theory) literatürde çok farklı disiplinlerin çalışma alanına girmektedir. Sosyolojiden matematiğe, işletmeden bilgisayar bilimlerine (Seker, 2015) (Arslan & Seker, 2014) kadar farklı alanlarda kullanılmaktadır (Seker, Altun, Ayan, & Mert, 2014). Örneğin bilgisayar bilimlerinin altında ayrık matematik (discrete math) veya endüstri mühendisliği altında yön eylem çalışmaları (operations research) veya matematiğin bir çalışma alanı olarak karşılaşılabilir (SEKER, 2015). Çizge teorisi temel olarak bir problemin kenar (edge) ve düğümler (node) ile modellenmesi ve bu modelin bir çizge şeklinde gösterilmesi ilkesine dayanmaktadır. Çizge teorisinde tanımlı olan bazı özellikler bu modelin çözümüne ve dolayısıyla gerçek problemin çözümüne yardımcı olmaktadırlar. Yani çizge teorisinin işe yaraması için öncelikle gerçek dünyadan bir problem çizge olarak modellenir, bu model çözülür ve daha sonra gerçek dünyaya uygulanır.

Çizge teoremindeki herhangi bir işlemden önce, çizgenin tanımı yapılmalıdır ve bu tanım aşağıdaki şekildedir:

$$G= (V,E)$$

Yukarıdaki bu tanım herhangi bir çizgenin (graph) düğümler (vertices) ve kenarlar (edges) kümesi olduğunu ifade etmektedir. Çoğu kaynakta bu ifade sıralı olarak kabul edilir yani öncelikle düğümler ardından da kenarlar gösterilmektedir (Biggs, Lloyd, & Wilson, 1986).

Çizgeleri, kenarların} yönlü olup olmamasına göre, yönlü çizgeler (directed graphs) ve yönsüz çizgeler (undirected graph) olarak ikiye ayırmak mümkündür. Ayrıca kenarların değer almasına göre değerli çizgeler (weighted graphs) veya değersiz çizgeler (unweighted graphs) isimleri verilebilir.



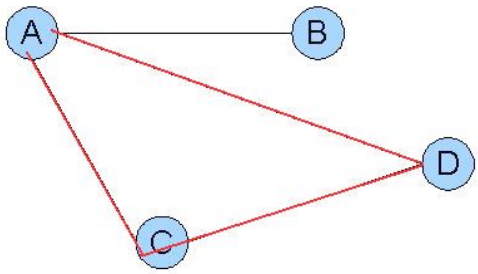
Örneğin yukarıdaki çizgede 4 düğümden ve 4 kenardan oluşan bir çizge gösterilmektedir. Bu çizgeyi

$$G= (\{A,B,C,D\} , \{(A,B),(A,C),(C,D),(A,D)\})$$

şeklinde ifade etmek mümkündür.

Yol (Path)

Bir çizge üzerinde bir veya daha fazla düğümden ve kenardan geçen rotaya verilen isimdir (Gibbons, 1985) . Örneğin aşağıdaki çizge üzerinde bir yol gösterilmiştir.



Yolların yazılışı ise geçtikleri düğümlerin sırasıyla yazılması ile elde edilir. Örneğin yukarıdaki yolu {A,C,D} olarak göstermek mümkündür.

Döngü (Cycle)

Çizge teorisinde bir düğümden başlayıp aynı düğümde biten yola (path) döngü adı verilir

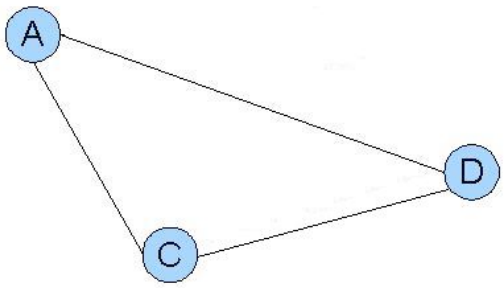
Örneğin yukarıdaki şekil 2'deki A düğümünden başlayarak yine bu düğümde biten {A,C,D} yoluna döngü denilebilir. Bir çizgenin içerisinde döngü bulunup bulunmadığının anlaşılması (cycle detection) problemi, literatürdeki önemli problemlerden birisidir ve yönsüz bir çizgedeki döngünün bulunma karmaşıklığı O(n) olacaktır (Sedgewick, 1983).

Güçlü Bağlı veya Tam Bağlı Çizgeler (Strongly Connected Graphs)

Bir çizgede bulunan bütün düğümleri diğer bütün düğümlere bağlayan birer kenar bulunuyorsa bu çizgeye güçlü bağlı çizge adı verilir. Döngü bulunma problemine benzer şekilde bir çizgenin güçlü bağlı çizge olup olmadığının anlaşılması veya bir alt çizgenin (çizgenin alt kümesi veya alt parçası olarak düşünülebilecek olan kısmı) içerisinde güçlü bağlı çizge olup olmadığının anlaşılması da önemli problemler arasında sayılabilir. Literatürde güçlü bağlı alt çizgelere özel olarak klik (clique) ismi verilir ve bu terim aynı zamanda sosyoloji gibi beşeri çalışma alanlarında bir toplulukta çok yakın olan arkadaş grubunu da ifade eder (örneğin bir okul veya bir iş yerindeki yakın arkadaş grubu). Bu tanım çizgenin her üyesinin (her düğüm) diğer üyeleri ile doğrudan ilişkide olduğu alt çizgeyi ifade etmektedir. Örneğin Kosaraju Algoritması, Tarjan Algoritması veya Dijkstra'nın güçlü bağlı çizge algoritması gibi çeşitli algoritmalar, bu amaçla bir çizgenin güçlü bağlı olup olmadığını bulmak için kullanılabilir (Cormen, Leiserson, Rivest, & Stei, 2001).

K-Düzenli Çizge (K-Regular Graph)

Bir çizge üzerindeki her düğümün "k" kadar komşusu bulunması durumuna k-düzenli çizge denilir (Chen). Örneğin aşağıdaki çizge 2-düzenli bir çizgedir çünkü her düğümün derecesi 2'dir.



Hamilton Yolu (Hamiltonian Path, Hamilton Circuit)

Bir yolun (path) Hamilton yolu olabilmesi için bir kere geçilen kenardan (edge) tekrar geçilmemesi gerekir ve ayrıca yolun bütün düğümleri (nodes) birer kere ziyaret etmesi gerekir.

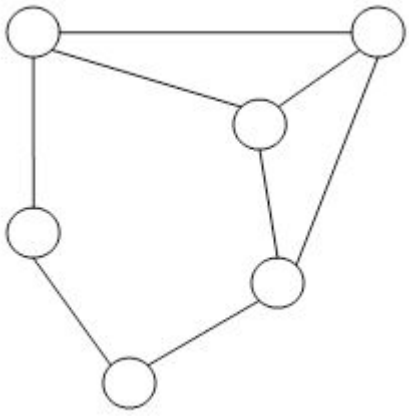
Bir Hamilton yolu başladığı yerde bitiyorsa yani tam bir döngü (cycle) tamamlıyorsa (yoldaki son düğümden ilk düğüme gidilmesi mümkünse) bu yollara Hamilton döngüsü (hamiltonian cycle) ismi de verilebilir (Garey & Johnson, 1979).

Hamiltonian yolları yönsüz çizgelerde (undirected graph) tanımlıdır ancak hamilton yollarının benzerleri yönlü çizgeler (directed graphs) için de uyarlanabilir.

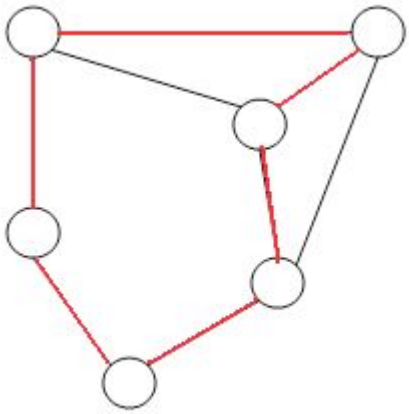
Bir çizgedeki, hamilton yollarının bulunması işlemi NP-Complete bir işlemdir.

Örnek

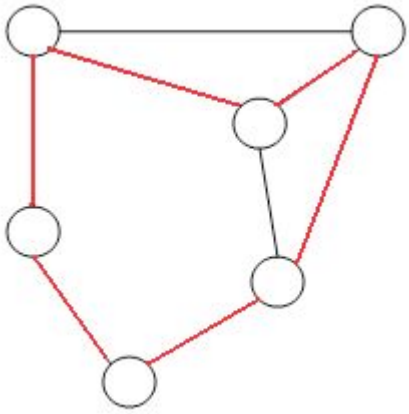
Örneğin aşağıdaki çizgeyi (graph) ele alalım:



Yukarıdaki şekil için tanımımıza uygun bir yol (path) elde etmek istersek:



Yukarıdaki şekilde kırmızı olarak gösterilen yol bir Hamiltonian yoludur. Ancak bu şekilde çıkarılabilecek tek yol yukarıdaki yol değildir.



Yukarıdaki bu yeni yol da tanıma uygun bir yoldur ve ilk çıkan yoldan farklıdır. Görüldüğü üzere bir çizgede tek bir Hamilton yolu bulunmak zorunda değildir.

Hamilton yollarının özellikleri

Herhangi bir hamilton döngüsü (hamiltonian cycle) tek bir kenarın (edge) çıkarılmasıyla bir hamilton yoluna (hamiltonian path) dönüştürülebilir.

2 den fazla düğüme (node) sahip ve tam bağlı (strongly connected) turnuva çizimleri birer hamilton yoludur (hamiltonian path)

Tam bağlı graflar (strongly connected graphs) için birbirinden farlı hamilton yollarının (hamiltonian path) sayısı (n-1)!/2 tanedir.

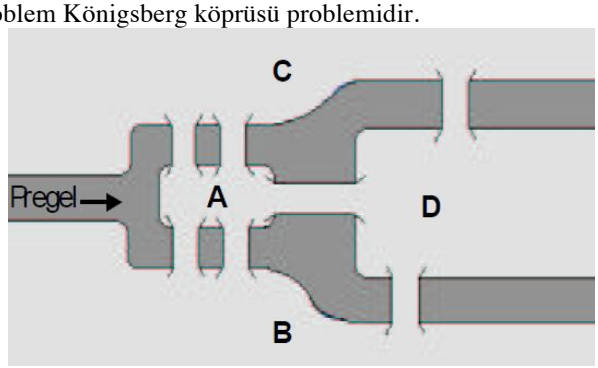
Hamiltonian yollarının kullanım alanları

Seyyar satıcı problemi (travelling salesman problem) gibi pek çok çizge teorisi problemin çözümünde kullanılabilirler.

Sıfır bilgi ispatı (zero-knowledge proof) gibi veri güvenliği (cryptography) problemlerinde kullanılabilirler.

Çizge Teorisinin Çıkışı ve Öyler Teorisi

Öylerin teorisini ortaya atmasında önemli rol oynayan tarihi problem Königsberg köprüsü problemidir.



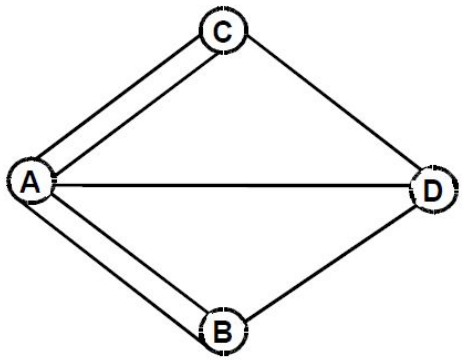
Yukarıdaki şekilde Pregel Nehri etrafında kurulu (C ve B karaları) ve nehrin ortasında iki adası olan (A ve D adacıkları) Kösigner şehrinin yukarıda görülen 7 köprüsü bulunmaktadır. Problem bütün köprülerden bir kere geçilen bir yol olup olmayacağıdır.

Öyler bu soruyla uğraşırken öyler yolu teorisini bulmuştur ve cevap olarak böyle bir yolun bulunamayacağını ispatlamıştır.

Öylerin iddiası basit bir keşfe dayanmaktaydı. Şayet bir düğüme (node) bir kenar (edge) ile geliniyorsa bu düğümü terk etmek için farklı bir yola ihtiyaç duyulur.

Bu durumda her düğümün derecesini (node order) hesaplayan Öyler, bir düğüme giren çıkan yolların sayısına düğüm derecesi (node order) ismini verdi.

Buna göre şayet bir düğümün derecesi tekse, bu düğüm ya başlangıç ya da bitiş düğümü olmalıdır. Bunun dışındaki durumlarda (yol (path) üzerindeki herhangi bir düğüm olması durumunda) tek sayıdaki yolun sonuncusu ziyaret edilmiş olamaz.



Yukarıdaki şekilde köprü örneğinin çizge ile gösterilmiş hali görülüyor. Burada dikkat edilirse her dört düğüm de tek sayıda dereceye sahiptir.

A 5

B C D ise 3

derecesine sahiptir. Bu durumda düğümlerden iki tanesi başlangıç ve bitiş olsa diğer iki tanesini birleştiren yollar kullanılamayacak ve bütün kenarlar gezilmiş olamayacaktır.

Öyler yolunun Tanımı

Öyler yolu (eulerian path) tam olarak şu şekilde tanımlanabilir:

Bir yönsüz çizgede (undirected graph) şayet bütün düğümleri (nodes) dolaşan bir yol bulunabiliyorsa bu yola Öyler yolu (Eulerian Path, Eulerian Trail, Eulerian Walk) ismi verilir. Bu yolun bulunduğu çizgeye ise yarı öyler (semi-eulerian) veya dolaşılabilir (traversable) çizge ismi verilir.

Şayet bu yolun başlangıç ve bitiş düğümleri (node) aynıysa bu durumda tam bir döngü (cycle) elde edilebiliyor demektir ve bulunan bu yola öyler döngüsü (eulerian cycle, eulerian circuit veya eulerian tour) ismi verilir (Mallows & Sloane, 1975). Bu yolu içeren çizge ise öyler çizgesi (eulerian graph veya unicursal) ismi verilir.

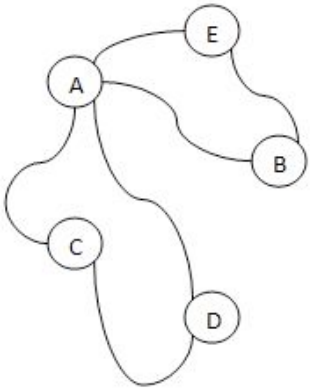
Yukarıdaki tanımı yönlü graflar (directed graphs) için de yapmak mümkündür. Ancak bu durumda yukarıdaki tanımda geçen yolları, yönlü yollar ve döngüleri, yönlü döngüler olarak değiştirmek gerekir.

Öyler yolunun özellikleri

- Bir yönsüz bağlı çizgenin bütün düğümlerinin derecesi çiftse bu çizge öyler çizgesidir (eulerian) [ancak ve ancak]
- Bir yönlü çizge (directed graf) ancak ve ancak bütün düğümlerin giren ve çıkan derecelerinin toplamları eşitse öyler çizgesi (eulerian) olabilir.
- Bir yönsüz çizgenin öyler yolu bulunabilmesi için iki veya sıfır sayıda tek düğüm derecesine sahip üyesi olmalıdır.

Öyler Döngülerinin sayısı

Bir çizgede öyler döngüleri birden fazla olabilir. Yani birbirinden farklı döngüler elde edilebilir. Burada fark oluşturan faktör başlangıç ve bitiş düğümleridir.



Örneğin yukarıdaki şekil için, A-B-E-A-C-D-A döngüsü bir öyler döngüsüdür. Benzer şekilde

E-B-A-C-D-A-E döngüsü de bir öyler döngüsüdür.

BEST Teoremi

Buradaki soru acaba bir çizgede kaç farklı öyler döngüsü olabilir?

Bu soruya cevap BEST teoremi ismi verilen ve teoremi bulan kişilerin isimlerinin baş harflerinden oluşsan teorem ile verilir. BEST teoremine göre bir çizgede bulunan öyler döngülerinin sayısı çizgedeki bütün düğümlerin derecelerinin bire eksiğinin faktöriyellerinin çarpımına eşittir (Tutte & Smith, 1941).

$$\prod (d(v)-1) !$$

olarak gösterilebilecek teoriye göre d() verilen düğümün (vertex) derecesi ve v ise çizgedeki bütün düğümlerdir.

Örneğin yukarıdaki çizge için bu değeri hesaplayacak olursak önce düğümlerin derecelerini çıkarmamız gerekir:

A 4

B 2

C 2

D 2

E 2

Şimdi bu değerlerin birer eksiklerinin faktöriyellerini çarpalım

3! = 6

1! = 1

1! = 1

1! = 1

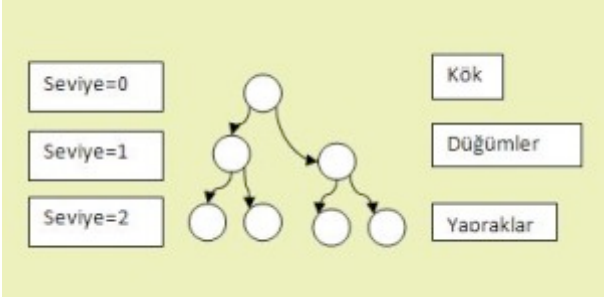
1! = 1

sonuç olarak 6x1x1x1x1 = 6 farklı öyler döngüsü bulunabilir diyebiliriz.

Yönlü Düz Ağaçlar (Directed Acyclic Graphs)

Bilgisayar bilimlerinde veri modellemede kullanılan Düz

ağaçların (acyclic graph), yani içinde herhangi bir döngü (daire) bulunmayan ağaçların, yani bir noktadan birden fazla geçme imkanı bulunmayan ağaçların, yön almış halleridir. Yani her kol (edge) bir yön göstermektedir ve gösterilen yönde ilerlemek mümkün iken tersi yönde ilerlemek mümkün değildir.



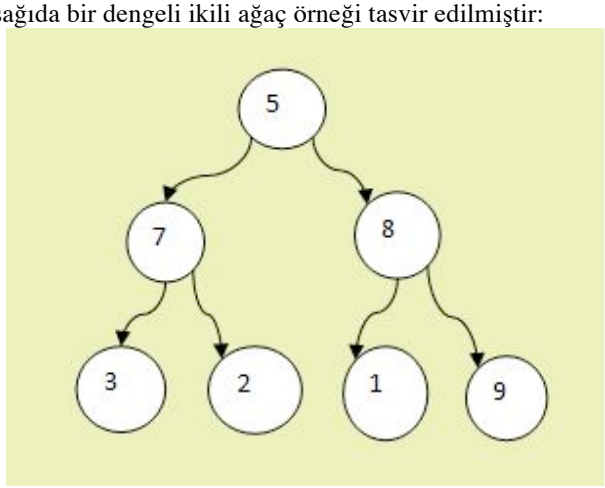
Yukarıdaki bu tanım aslında bir ağaç (tree) tanımının aynısıdır. Bu yüzden temel bir ağaç aslında bir yönlü düz ağaç olarak nitelendirilir.

Yukarıdaki ağaç tasvirinde 7 düğümden (node) oluşan ve yapraklarında (leaf) 4 düğüm bulunan bir ağaç gösterilmiştir. Bu ağacın derinliği (depth) 2 dir ve her seviyenin (level) değeri yanında verilmiştir. Ağaçların 1 tane başlangıç düğümü bulunur ve bu başlangıç düğümüne kök (root) denilir.

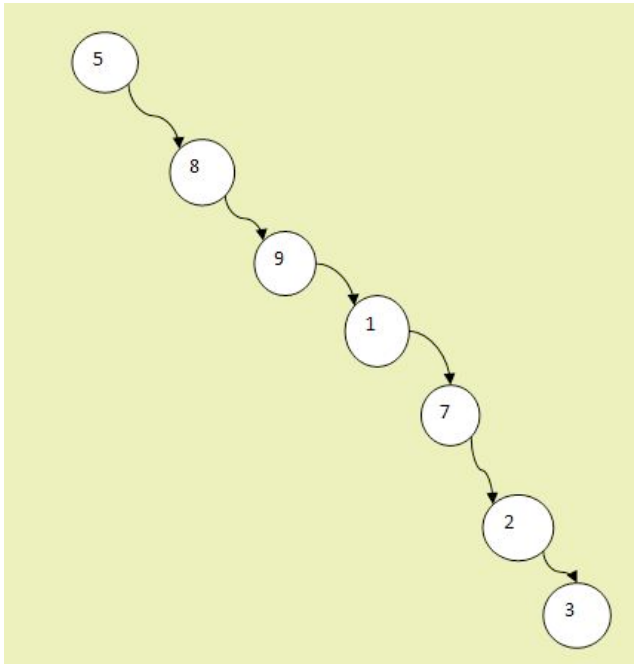
Özel olarak yukarıdaki ağacın her düğümünden sadece ikişer alt düğüme bağlantı bulunduğu için bu ağaca ikili ağaç (binary tree) adı da verilebilir.

İkili Ağaçlar (Binary Tree)

Ağaçların özel bir hali olan ikili ağaçlarda her düğümün çocuklarının sayısı azami 2 olabilir. Bir düğümün daha az çocuğu bulunması durumunda (0 veya 1) ağacın yapısı bozulmaz. Yapraklar hariç bütün düğümlerin ikişer çocuğu bulunması ve yaprakların aynı derinlikte bulunması durumunda bu ağaca dengeli ağaç (balanced tree) denilir. Aşağıda bir dengeli ikili ağaç örneği tasvir edilmiştir:



Bu ağacı değişik sıralarda yeniden oluşturabiliriz. Örneğin aşağıdaki ağaç da yukarıdaki verilerin aynılarını taşıyan bir ikili ağaç örneğidir.

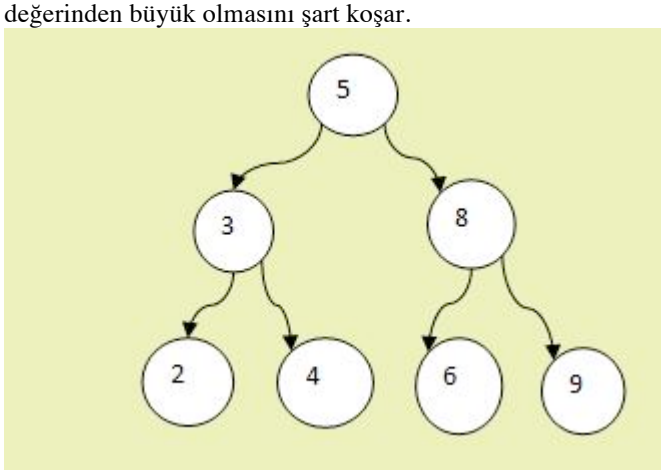


Yukarıdaki bu ağacın ilk örnekten farkı dengesiz olması ve özel olarak her düğümün çocuk sayısının 1 olmasıdır. Tanım hatırlanacak olursa yukarıdaki bu ağaç da bir ikili ağaç olarak kabul edilebilir.

İkili Arama Ağacı (Binary Search Tree)

İkili ağaçların (Binary Tree) özel bir hali olan ikili arama ağaçlarında, düğümlerde duran bilgilerin birbirine göre küçüklük büyüklük ilişkisi bulunmalıdır. Örneğin tam sayılardan (integer) oluşan veriler tutulacaksa bu verilerin aralarında küçük-büyük ilişkisi bulunmaktadır.

İkili arama ağacı, her düğümün solundaki koldan ulaşılabilecek bütün verilerin düğümün değerinden küçük, sağ kolundan ulaşılabilecek verilerin değerinin o düğümün değerinden büyük olmasını şart koşar.



Örneğin yukarıda bir ikili arama ağacı tasvir edilmiştir. Bu ağaçta dikkat edilecek olursa kökün solunda bulunan bütün sayılar kökten küçük ve sağında bulunan bütün sayılar kökten büyüktür.

İkili arama ağacına bu kurala uygun olarak ekleme çıkarma veya silme işlemleri yapılabilir. Ancak yapılan her işlemden sonra ikili arama ağacının yapısı bozulmamış olmalıdır.

İkili arama ağacında arama işlemi:

Yukarıda da anlatıldığı üzere ağacın üzerinde duran verilerin özel bir şekilde sıralı bulunması gerekir. Buna göre herhangi bir değeri arayan kişi ağacın kök değerine bakarak aşağıdaki 3 eylemden birisini yapar:

Aranan sayı kökteki değere eşittir -> Aranan değer bulunmuştur

Aranan sayı kökteki değerden küçüktür -> Kökün sol kolu takip edilir

Aranan sayı kökteki deüerden büyüktür -> Kökün sağ kolu

takip edilir

İkili arama ağaçlarının önemli bir özelliği ise öz çağıran(recursive) oluşudur. Buna göre bir ağacın sol kolu veya sağ kolu da birer ağaçtır. Bu özellikten faydalanarak örneğin C dilinde aşağıdaki şekilde bir arama fonksiyonu yazılabilir:

```
dugum * ara(dugum *kok, int deger){
     if(deger == kok->veri){
        return kok;
    }
    else if(deger > kok->veri ){
        return ara(kok->sag, deger);
    }
    else{
        return ara(kok->sol,deger);
   }
}
```

Yukarıdaki bu örnekte daha önce ikili ağaçlar konusunda tanımlanmış bir düğüm struct'ı kullanılmıştır. Koda dikkat edilecek olursa özçağıran bir yapı görülebilir. Yani fonksiyonun içerisinde kendisi çağrılmış ve yeni kök değeri olarak mevcut kökün sol veya sağ değeri verilmiştir. Arana değer her durumda değişmediği için alt düğümlere kadar aynı olarak indirilmiştir.

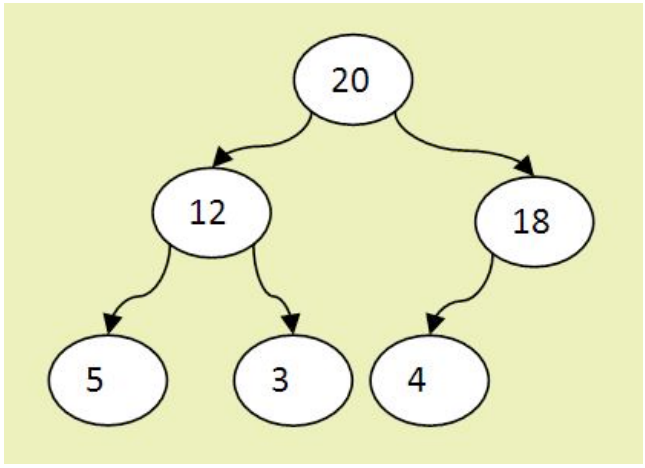
Yukarıdaki koddaki önemli bir eksiklik ise kodun ağaç üzerinde bir sayıyı bulamaması durumunun tasarlanmamış olmasıdır. Bu durumda basitçe NULL değer kontrolü ile aşılabilir. Buna göre sol veya sağa gitmeden önce sol veya sağda bir değer olduğuna bakılmalı şayet değer varsa gidilmelidir. Şayet değer yoksa ağaçta bu sayının bulunma ihtimali yoktur ve bu sayının olmadığını belirten bir ifade (örneğin NULL) döndürülmelidir. Kodun bu şekilde tashih edilmiş hali aşağıda verilmiştir:

```
dugum * ara(dugum *kok, int deger){
     if(deger == kok->veri){
        return kok;
    }
    else if(deger > kok->veri ){
      if(kok->sag != NULL)
             return ara(kok->sag, deger);
      else
            return NULL;
    }
    else{
      if(kok->sag != NULL)
              return ara(kok->sol,deger);
      else
            return NULL
   }
}
```

Yığın Ağacı (Heap Tree)

Yığın ağacı bilgisayar bilimlerinde özellikle sıralama amacıyla çokça kullanılan bir veri yapısıdır. Bu veri yapısı üst düğümün (atasının) alt düğümlerden (çocuklarından) her zaman büyük olduğu bir ikili ağaç (binary tree) şeklinde düşünülebilir.

Aşağıda örnek bir yığın ağacı verilmiştir:



Yukarıdaki ikili ağaçta dikkat edilirse ağaç dengeli bir şekilde sırayla doldurulmuştur. Buna göre yeni bir eleman daha eklenmesi durumunda ağacın eleman sayısı 7'ye yükselecek ve sayıların yeri değişmekle birlikte yeni eleman şu andaki 18 sayısını içeren düğümün sağına gelecektir.

Ayrıca herhangi bir ağacı yığın ağacına çevirmek (yığınlamak, heapify) de mümkündür ve bu işlem bir yığın ağacı oluşturma işleminin temelini oluşturur. Basitçe yığın ağacına yapılan her ekleme ve her çıkarma işleminden sonra bu yığınlama (heapify) işlemi yapılarak yığın ağacının formunu koruması sağlanmalıdır.

Yığın Sıralaması (Heap Sort)

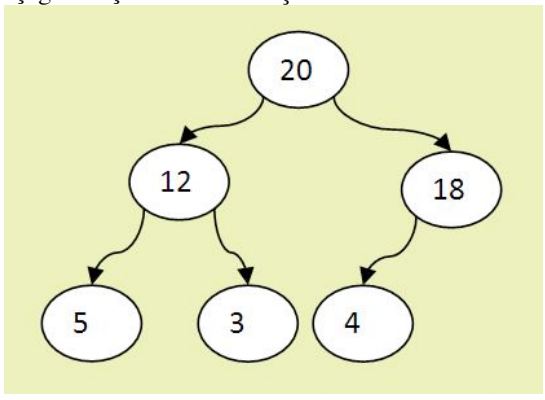
Verinin hafızada sıralı tutulması için geliştirilen sıralama algoritmalarından (sorting algorithms) bir tanesidir. Yığınlama sıralaması, arka planda bir yığın ağacı(heap) oluşturur ve bu ağacın en üstündeki sayıyı alarak sıralama işlemi yapar. (Lütfen yığın ağacındaki en büyük sayının her zaman en üstte duracağını hatırlayınız.)

Sıralanmak istenen verimiz:

5,12,20,18,4,3

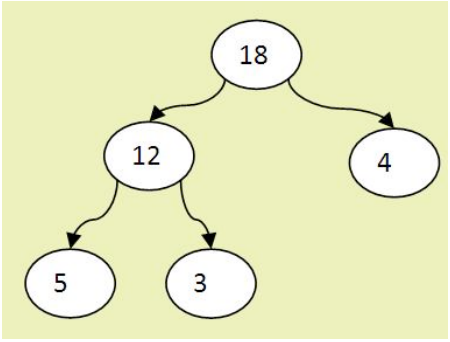
olsun. Bu verilerin bir oluşumun(composition) belirleyici alanları olduğunu düşünebiliriz. Yani örneğin vatandaşlık numarası veya öğrenci numarası gibi. Dolayısıyla örneğin öğrencilerin numaralarına göre sıralanması durumunda kullanılabilir.

Yukarıdaki bu sayılardan bir yığın ağacı oluşturulduğunda aşağıdaki şekilde bir sonuç elde edilir:

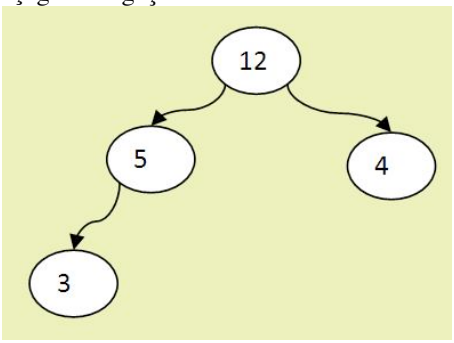


Bu ağacın en büyük değeri en üstte (kökte) durmaktadır öyleyse bu değer alınarak sonuç dizisinin son elemanı yapılırsa ve sonra geriye kalan sayılar tekrar yığınlanırsa (heapify) ve bu işlem eleman kalmayana kadar tekrarlanırsa sonuç dizisindeki veriler sıralanmış olarak elde edilir. Bu işlemler adım adım aşağıda gösterilmiştir:

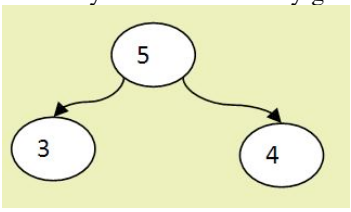
Sonuç dizisi : {20}



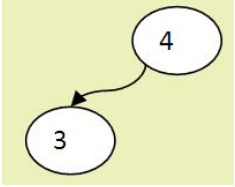
20 sayısı yığın ağacından silindikten sonra kalan sayılar yığınlandıktan sonra yukarıdaki ağaç elde edilir. Bu ağacın en üstündeki sayı alınırsa sonuç dizisi : {20,18} olarak bulunur ve 18 alındıktan sonra kalan sayılar yığınlandığında ise aşağıdaki ağaç elde edilir.



Bu ağacında en üstünde bulunan sayı 12'dir ve bu sayı sonuç dizisine eklenir. Sonuç dizisi : {20,18,12} olarak bulunur ve kalan sayılar bir kere daha yığınlanır:



Sonuçta kalan sayıların yığınlanmasıyla yukarıdaki ağaç elde edilir ve bir kere daha en üstteki sayı alınarak sonuç dizisi : {20,18,12,5} olarak bulunur.



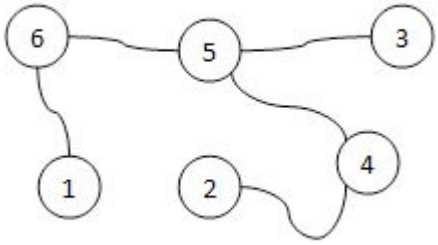
Aynı işlem kalan sayılara tekrar tekrar uygulanarak {20,18,12,5,4,3} sayıları elde edilir. Dikkat edilirse bu sayılar ilk başta verilen sayıların sıralanmış halidir. Şayet sıralama işlemi küçükten büyüğe yapılsın isteniyorsa bu durumda sayılar dizinin başından değil sonundan yazılarak kaydedilebilir veya çıkan sonuç tersten sıralanabilir.

Prüfer Dizilmi (Prüfer Sequence)

Ağaç (tree) yapısındaki graflar için yani dairesel olmayan çizgeler (acyclic graphs) için kullanılabilir. Daha basit bir ifade ile çizgenin (graph) yaprakları (leaf) bulunmalıdır.

Prüfer diziliminin altında yatan sorun bir ağacın bir sayı dizisi ile nasıl gösterileceğidir. Yani bir çizgede birbirine bağlı düğümler (nodes) olduğunu ve bu çizge yapısını bir şekilde sayılarla göstermek (veya dizi (array)) istediğimizi düşünelim.

Örneğin aşağıdaki çizgeyi (graph) ele alalım:



şekildeki çizge toplam 11 düğüm (node) bulunmaktadır ve şekilde çizge yönsüz çizgedir (undirected graph) aynı zamanda da döngü (cycle) bulundurmaz.

Bu durumda yukarıdaki şekli prüfer dizilimi olarak göstermek mümkündür.

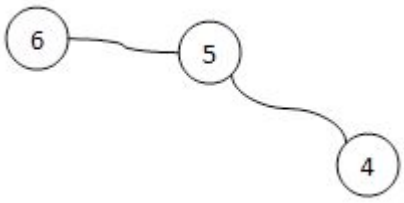
Algoritmamız son iki düğüm kalana kadar çalışır. Bu durumda önce yapraklardan başlanarak işlem yapıyoruz. Yukarıdaki çizgeyi yapraklardan köke doğru artan sayılar ile bu yüzden numaralandırdık.

Sırasıyla her düğümün bağlı olduğu diğer üst düğümü yazıyoruz.

Bu durumda aşağıdaki düğümler yapraklardır ve bağlı oldukları üst düğümler yanlarında verilmiştir:

1 → 6
2 → 4
3 → 5

Yukarıdaki yapraklardan başlayarak bu dizilimi not ediyoruz. Yani şimdilik prüfer dizilimimizde 645 bulunuyor. İkinci adımda bu yaprakları çizgeden (graph) kaldırıyoruz.



Şeklin yeni halindeki yaprakları not edeceğiz ancak son iki düğüm kalınca duracağımız için yeni şekilden sadece bir düğümü alıyoruz. Bu düğümde numara sırasına göre (tamamen tesadüfen) 4 oluyor

4 → 5

şeklinde son sayımızı da not ettikten sonra prüfer dizilimimizdeki sayılar 6455 olarak kaydedilmiş oluyor.

Artık iddia edebiliriz ki 6455 sayı serisinden (sequence) 6 ve 5 düğümlerini bilerek yukarıdaki çizge (graph) geri çizebiliriz.

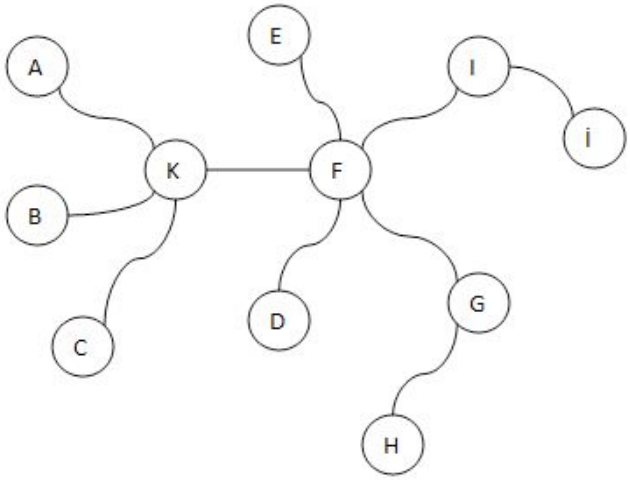
Bu iddianın tatbiki oldukça basittir. 6 ve 5'ten oluşan bir çizge çizilir ve ardından prüfer dizilimi tersten okunarak sırasıyla yeni düğümler (nodes) çizgeye eklenir.

İki Parçalı Çizgeler (Bipartite Graphs)

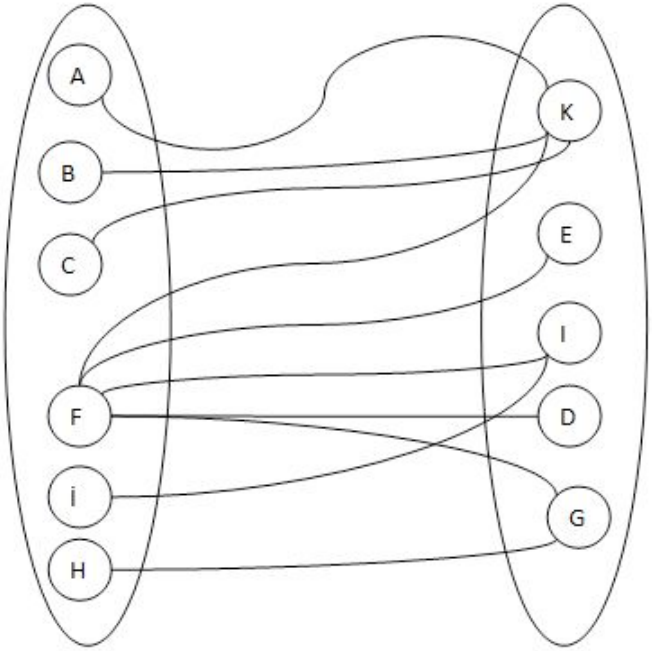
Bir çizgeyi oluşturan düğümleri iki farklı kümeye ayırabiliyorsak ve bu iki kümenin elemanlarından küme içerisindeki bir elemana gidilmiyorsa. Yani bütün kenarlar (edges) kümeler arasındaki elemanlar arasındaysa, bu çizgeye iki parçalı çizge (bipartite graph) ismi verilir.

İki parçalı çizgelere örnekler

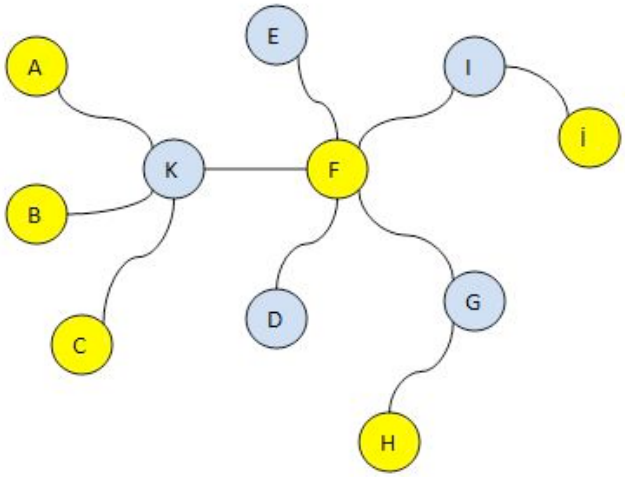
Örneğin aşağıdaki şekilde gösterilen çizgeyi iki parçalı olup olamayacağını incelemeye çalışalım:



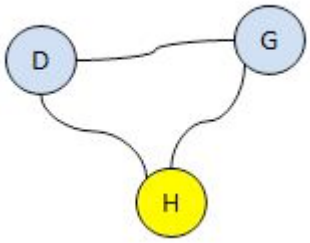
Yukarıda örnek olarak verilen çizgenin düğümlerini koşulumuzu sağlayacak şekilde yani iki ayrı küme oluşturacak ve bu kümelerin içerisinde yol bulunmayacak şekilde yerleştirmeye çalışalım.



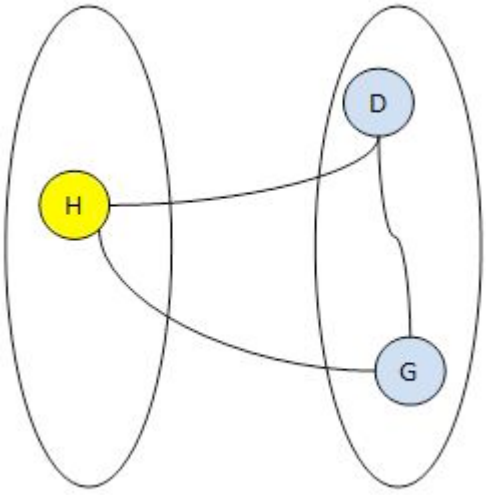
Yukarıdaki şekilde, ilk şekilden farklı olarak sadece düğümlerin (nodes) yerleri değiştirilmiştir. Yukarıdaki şekilde ayrıca iki ayrı küme oluşturulmuş ve ilk küme ile ikinci küme arasında bölünen düğümlerin tamamında kenarlar karşı kümeye işaret etmiştir. Yani aynı küme içerisinde hiçbir kenar bulunmamaktadır.
Bu durumu aşağıdaki şekilde de gösterebilirdik:



Yukarıdaki şekilde görüldüğü üzere komşu olmayan düğümler renklendirilmiştir. Yani komşu iki düğüm aynı renkle renklendirilmemiştir. Daha farklı bir deyimle bir düğümün komşuluk listesindeki (adjacency list) bütün düğümler aynı renkte boyanmıştır. Sonuçta aynı renkte boyanan iki komşu oluşmamaktadır.
Aşağıdaki iki parçalı olmayan çizgeye de aynı şekilde yaklaşırsak problem ortaya çıkacaktır.



Yukarıdaki şekil bipartite tree (iki parçalı ağaç) değildir. Çünkü şekildeki D ve G aynı renktedir. Bu düğümlerden birisinin rengi değiştirildiğinde bu sefer de H ile aynı renkte olacaktır. Dolayısıyla hiçbir şekilde iki grup elde edilemez. Farklı bir ifadeyle:

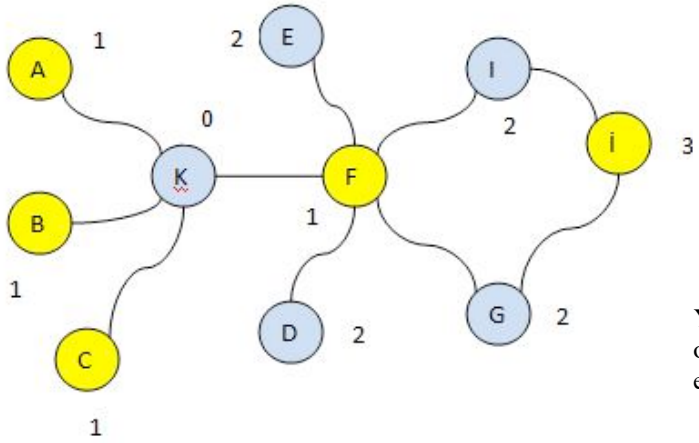


Çizgeyi yukarıdaki şekilde iki gruba bölmek istersek görüldüğü üzere aynı gruptaki iki düğüm arasında bir kenar bulunmakta bu durumda da iki parçalı ağaç olamamaktadır.

İki parçalı çizgenin test edilmesi

Bir çizgenin iki parçalı olup olmadığını (bipartite) test etmek için ne yazık ki yukarıda gösterildiği gibi çizgedeki bütün düğümlerin yerlerinin değiştirilmesi, iki kümede toplanması ve sonunda da aralarında bir kenar olup olmadığına bakılması

mümkün değildir.
Bir çizgenin iki parçalı olduğunu anlamak için çizgenin bütün elemanları arasında tek ve çift ayrımı yapılabilir. Örneğin rast gele seçilen bir düğümden başlanarak diğer bütün düğümlerin bu düğüme olan mesafesini yazmamız ve neticesinde tek ve çift düğümlerin komşuluğunu kontrol etmemiz yeterlidir. Yani çift veya tek iki düğüm birbirine komşu değilse bu çizgeye iki parçalı çizge ismi verilebilir.



Yukarıdaki şekilde başlangıç için rast gele olarak 0 seçilmiş olsun. Bu çizgede K düğümüne olan uzaklıklarına göre bütün düğümlere mesafeleri yazılmıştır. Örneğin İ düğümü K düğümüne 3 düğüm uzaklıktadır. Sonrada bütün düğümler tek ve çift olmasına göre renklendirilmiştir. Yani tek sayıdaki uzaklıkta olanlar sarı, çift sayıdaki uzaklıkta olanları ise mavi boyanmıştır. Bu boyama sadece görüntüleme için kullanılmıştır.
Programlama sırasında bilgisayar tek ve çift sayıdaki düğümleri kontrol edebilir. Örneğin G ve İ düğümleri komşudur. Benzer şekilde İ ve I düğümleri de komşudur. Şayet bu düğümler bir şekilde aynı şekilde olsaydı (ikiside çift veya ikisi de tek olsaydı) bu durumda iki parçalı bir grafik değildir sonucuna varılacaktı.

İki parçalı Çizgelerin kullanım alanları

Özellikle eşleştirme problemleri için oldukça kullanışlıdırlar. Eşleştirme problemleri (matching problems) genelde iş/işçi eşleştirmesi, evlilik, problem / çözüm eşleştirmesi gibi farklı unsurları birleştirmek için kullanılırlar.
Örneğin k kişisi i işi için uygunsa k kişisi ile i işi arasında bir kenar bulunuyor demektir. Bu iş ve kişilerden oluşan grafikte atama hatalarının olup olmadığının bulunması iki parçalı ağaç bulan algoritmalar için oldukça basittir.
Örneğin paralel işleme (eş zamanlı işleme) ve koşut zamanlı işleme (concurrent processing) gibi aynı anda birden fazla işin beraber yapıldığı işlerde kullanılan petri ağları (Petri nets, place / transition nets) gibi ağların modellenmesinde ve çalıştırılmasında önemli rol oynarlar.

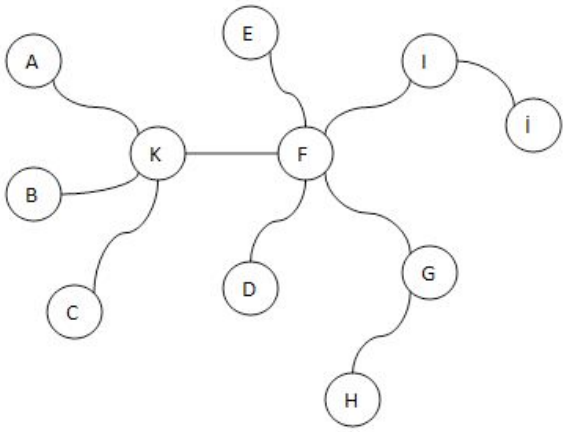
iki parçalı Çizgelerin özellikleri

iki parçalı çizgeler için aşağıdaki çıkarımlar ve koşullar sıralanabilir:

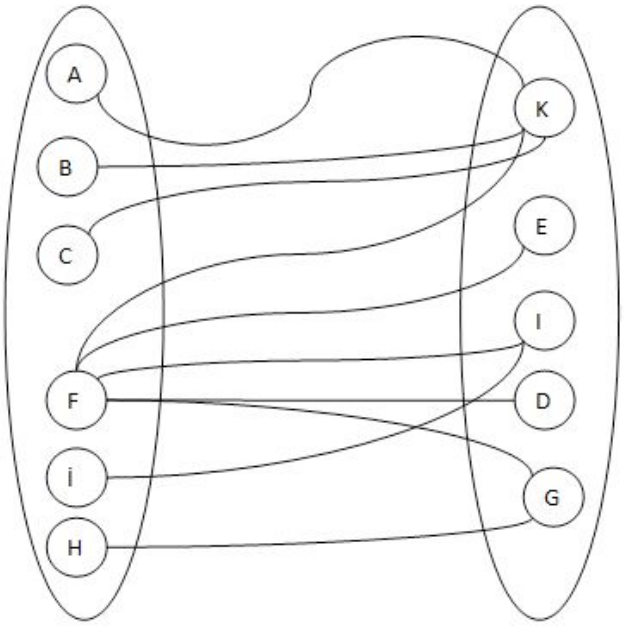
- Bütün ağaçlar iki parçalı çizgedir
- Şayet bir çizge döngü (cycle) içermiyorsa iki parçalı çizgedir
- Şayet bir çizgede tek sayıda düğümden oluşan bir döngü (cycle) içeriyorsa iki parçalı çizge değildir.
- Şayet bir çizge boyandığında iki renk veya daha az renkle boyana biliyorsa bu çizge iki parçalı çizgedir.

Denkşekillilik (Isomorphic)

İki şeklin birbirinden farklı ancak denk olması durumudur.
Bilgisayar bilimleri de dahil olmak üzere pek çok bilim ve mühendislik alanında kullanılan çizge teorisine (graph theory) göre iki şekil birbirinden farklı çizilmiş ancak işlev ve değer olarak aynı olabilir.
Tanım ve örnek
Örneğin aşağıdaki iki şekli ele alalım:



Yukarıda verilen çizgedeki düğümler (nodes) iki farklı küme oluşturmak için aşağıdaki şekilde yerlerinden hareket ettirilmiş olsun:



Yukarıdaki bu yeni şekil ve ilk şekil arasında denklik söz konusudur. Buradaki denklik komşuluk anlamında düşünülebilir. Yani iki şekilde de bütün düğümlerin komşuları aynıdır. Ancak fark edileceği üzere iki şekildeki elamanların yerleri değiştirilmiştir.
Bu durumda yukarıdaki iki şekil için denkşekilli (isomorphic, izomorfik) denilebilir.
Yukarıdaki tanım için aynı zamanda kenar korumalı (edge preserving) birebir ve örten dönüşüm (bijection) terimi kullanılabilir.

Denkeşliliğin özellikleri

Şayet iki çizge denkeşli ise bu çizgelerin birisindeki döngü (cycle) sayısı diğerine eşittir.
Şayet iki çizge denkeşli ise bu çizgelerden birisindeki toplam düğüm dereceleri (node order) diğerine eşittir.

Denkeşlilik Problemi (Isomorphism problem)

Denkeşlilik ile ilgili önemli bir problem iki grafın birbirine denk olduğunun tespitinde yaşanır. Elimizde iki farklı graf olduğunu ve bunların denkeşli (isomoprhic) olup olmadığını öğrenmek istediğimizi düşünelim.
Herşeyden önce bu problemi ilginç yapan, problemin polinom zamanda çözülüp çözülemeyeceğidir. Bu problem bu anlamda ilginçtir çünkü problemi ne NP ne de P kümesine ait değildir denilebilir.

Aslında bu tartışma daha derinlerde NP ☐ P iddiasının doğruluğuna dayanmaktadır. Yani şayet NP, P nin üst kümesiyse (superset) bu durumda ikisi arasında başka bir küme olamaz. Ancak bazı araştırmacılar (örneğin Uwe Schöning) bu iki küme arasında alt ve üst hiyerarşik seviyeler bulunduğunu iddia etmişler ve denkşekillik gibi bazı problemleri bu kümeye dahil etmişlerdir.

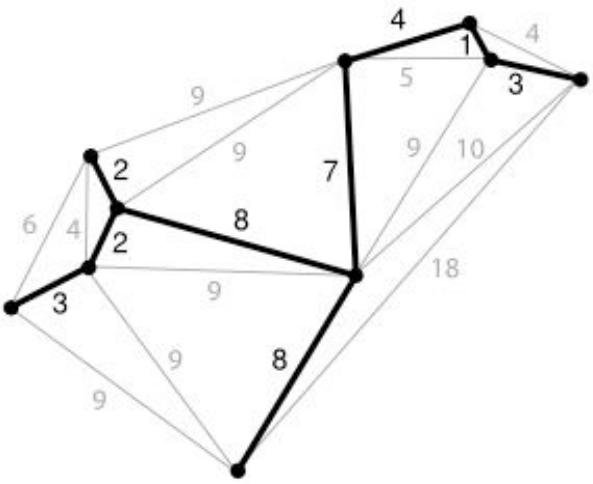
İddiaya göre P altta ve NP üstte iki seviye olarak düşünülürse düşük hiyerarşik seviyedeki problemler P'ye ve yüksek hiyerarşik seviyedeki problemler NP'ye daha yakın kabul edilecektir.

Bütün bu iddiaların sebebi denkşekillik gibi problemlerin karmaşıklığının (complexity) hala ispatlanamamış olmasıdır.

Bu problemin farklı bir halinin karmaşıklığı bulunabilmektedir. Örneğin problemimizi biraz değiştirerek alt grafların (subgraphs) denkşekilliğini sorgulayacak olursak bu durumda problemimiz NP-Complete olarak sınıflandırılabilir.

Asgari Tarama Ağacı (Minimum Spanning Tree)

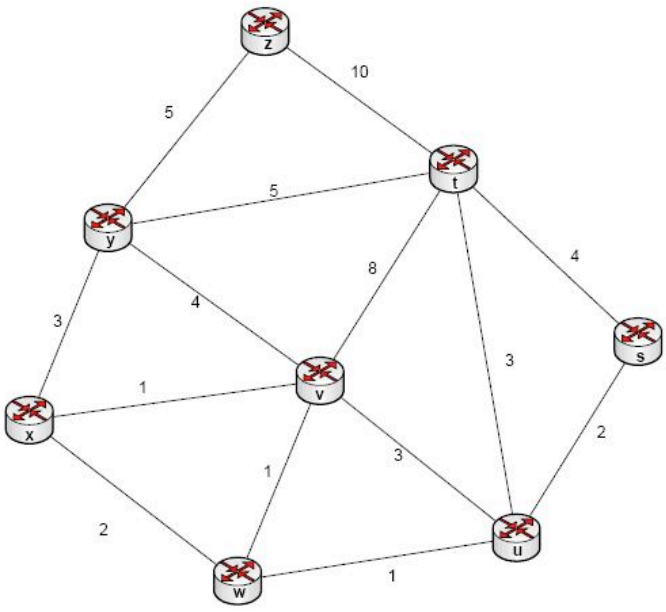
Asgarai tarama ağacı, ağırlıklık bir ağda (weighted graph, yani her düğümü birbirine bağlayan yolların maliyeti (ağırlığı) olması durumu), bütün düğümleri dolaşan en kısa yolu verir. Örneğim aşağıdaki grafikte bütün düğümlere uğrayan en kısa yol işaretlenmiştir:



asgari tarama ağacını veren en meşhur algoritmalar

Kruskal Asgari Tarama Ağacı Algoritması

Kruskal algoritmasının çalışması bir örnek üzerinden açıklanmıştır.



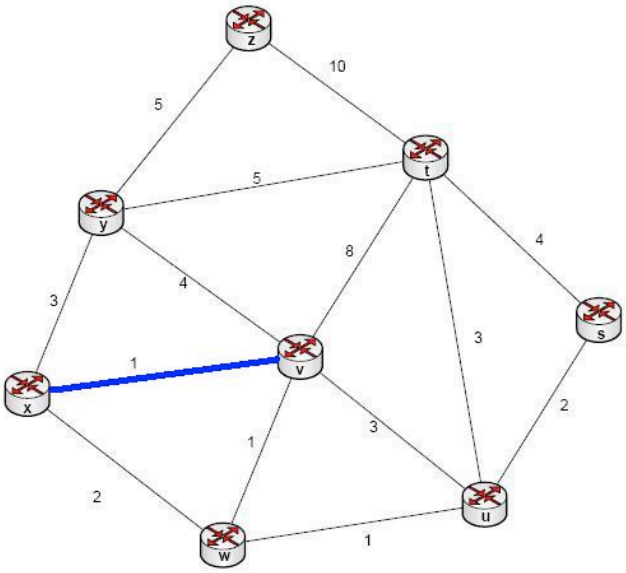
Yukarıdaki grafikte her düğüm için bir temsili harf ve her bağlantı için bir ağırlık değeri atanmıştır. Buna göre her düğümden diğerine gitmenin maliyeti belirlenmiştir.

Kruskal algoritmasında bütün yollar listelenip küçükten büyüğe doğru sıralanır. Bu liste yukarıdaki grafik için aşağıda verilmiştir:

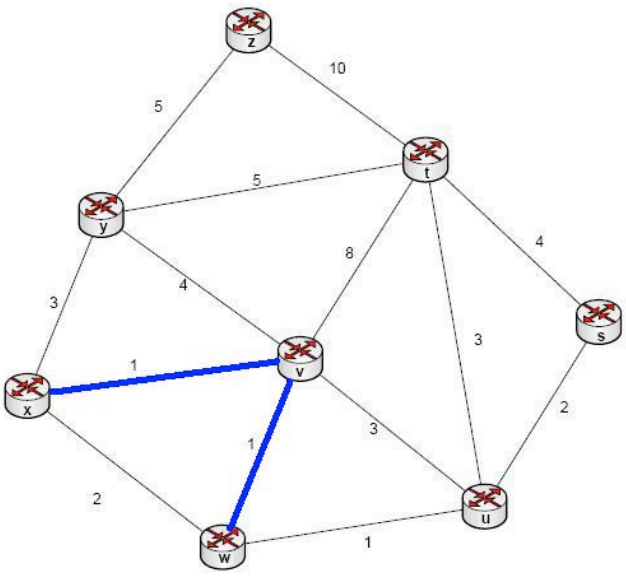
x-v:1
w-v:1
w-u:1
x-w:2
u-s:2
x-y:3
t-u:3
u-v:3
y-v:4
s-t:4
y-z:5
y-t:5
z-t:10

Yukarıdaki liste çıkarıldıktan sonra sırasıyla en küçükten en büyüğe doğru komşuluklar işaretlenir. Bu işaretleme sırasında ada grupları ve grupların birbiri ile ilişkisine dikkat edilir. Yani şayet listedeki iki düğüm harfi de aynı adadan ise bu bağlantı atlanır. Aşağıda sırasıyla bu grafikteki adaların oluşması ve asgari tarama ağacının çıkarılması gösterilmiştir:

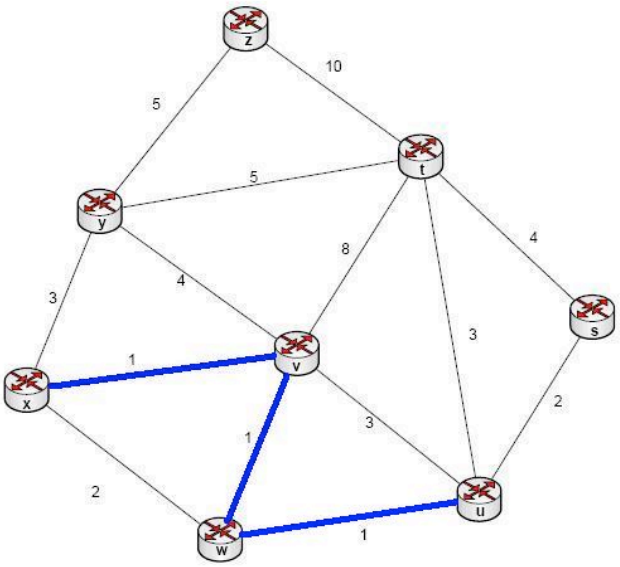
x-v:1



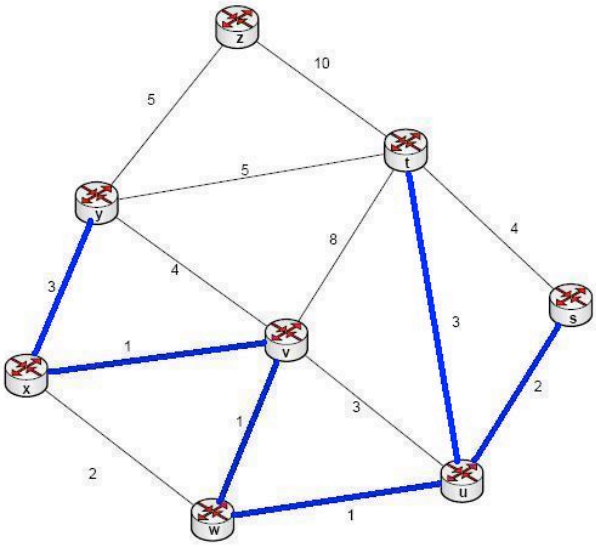
w-v:1



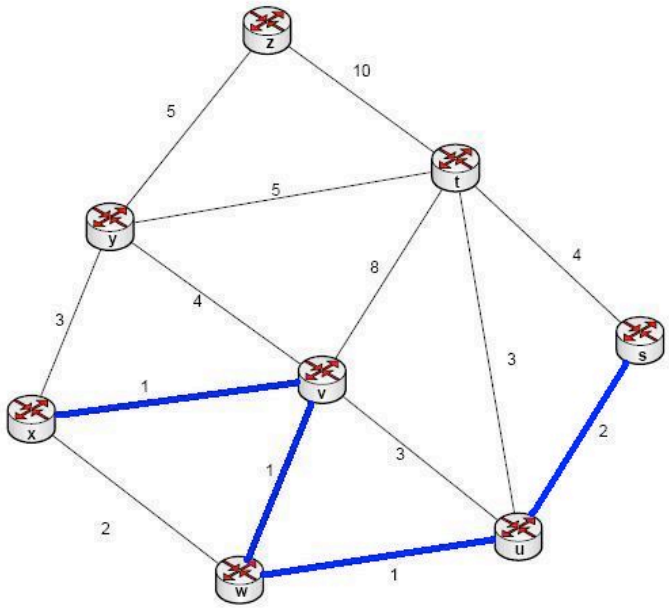
w-u:1



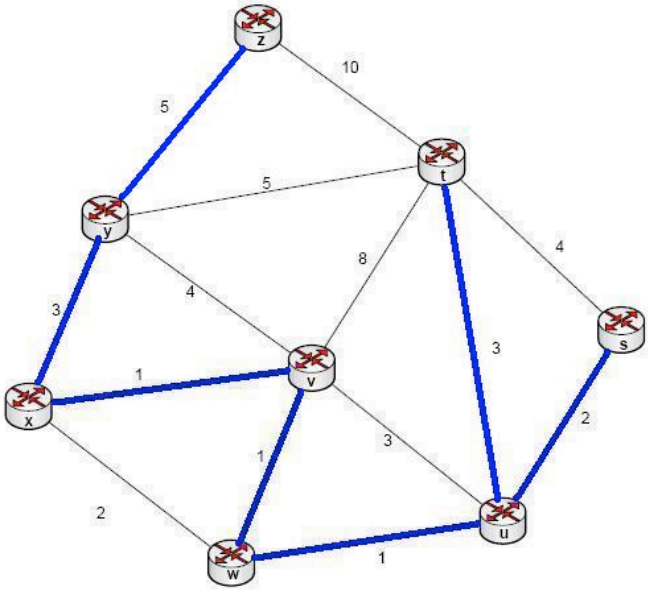
x-w bağlantısı atlanır, çünkü iki düğüm de zaten dolaşılmıştır bunun yerine u-s:2 bağlantısına atlanır.



Bu noktadan sonra u-v:3 , y-v:4 , s-t:4 , y-z:5 bağlantılarındaki her iki düğümde aynı adada olduğu için atlanır ve y-t:5 bağlantısına geçilir.



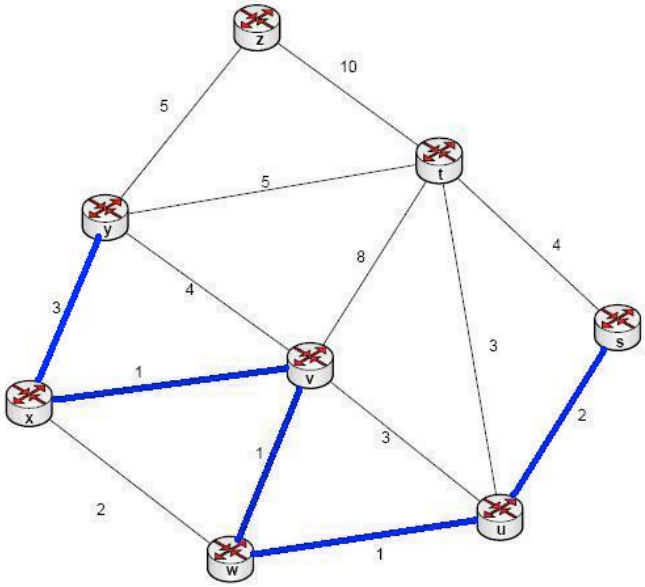
x-y:3



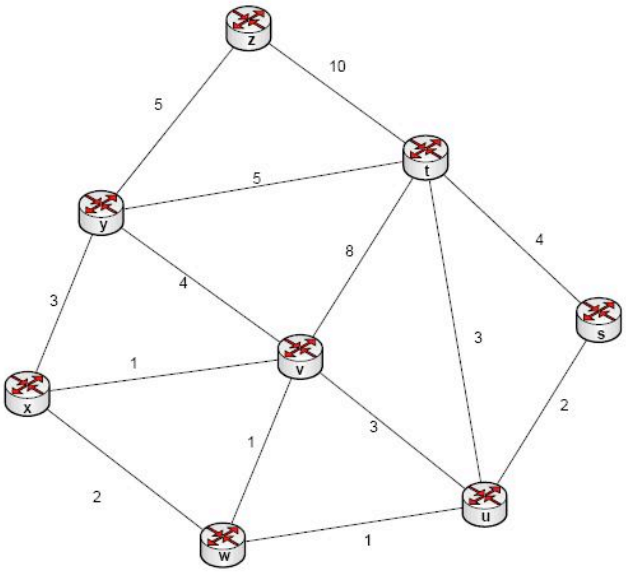
z-t:10 bağlantısı ise iki düğüm de gezildiği için yine gereksizdir.

Prims Asgari Tarama Ağacı

Bir asgari tarama ağacı (minimum spanning tree) algoritması olan Prim algoritması, işaretlemiş olduğu komşuluklara en yakın düğümü bünyesine katarak ilerler. Buna göre aşağıdaki grafiğin asgari tarama ağacını çıkaralım:



t-u:3



Yukarıdaki grafikte her düğüm için bir temsili harf ve her bağlantı için bir ağırlık değeri atanmıştır. Buna göre her düğümden diğerine gitmenin maliyeti belirlenmiştir.

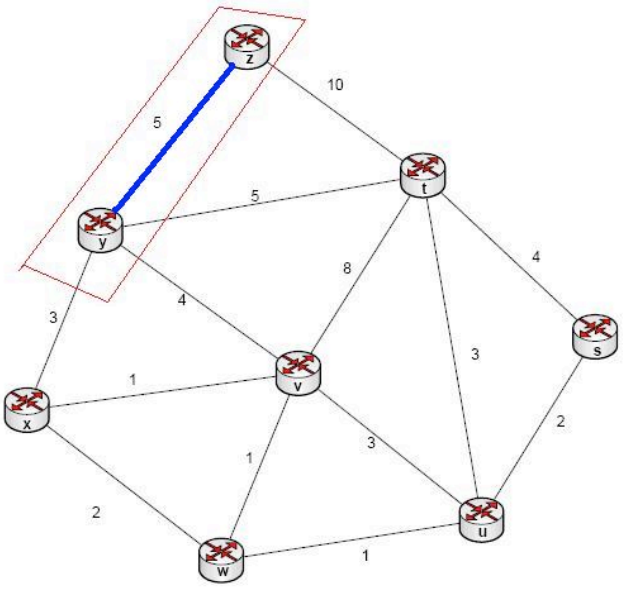
Prim algoritmasında rasgele bir başlangıç noktası seçilir. Örneğin bizim başlangıç noktamız "z" düğümü olsun. Bu durumda ilk inceleyeceğimiz komşuluk, "z" düğümünden gidilebilen düğümler ve maliyetleri olacaktır.

z düğümünden gidilebilen düğümler ve maliyetleri aşağıda listelenmiştir:

y:5

t:10

Prim algoritması bu listedeki en küçük maliyetli komşuyu bünyesine dahil eder. Buna göre yeni üyelerimiz {z,y} olacaktır ve gidilen yollar {z-y:5} olacaktır. (ilk üyeler listesinde şimdiye kadar ziyaret edilmiş düğümler bulunur. Bu düğümler listesinde zaten olan bir düğüm listeye eklenemez. yollar listesinde ise hangi düğümden hangi düğüme ne kadar maliyetle gidildiği tutulur.) Dolayısıyla grafiğimizde Prim algoritması tarafından işaretlenen düğümler aşağıda gösterilmiştir:



şimdi üyelerimizin durduğu listedeki bütün düğümlerin komşularını listeleyelim:

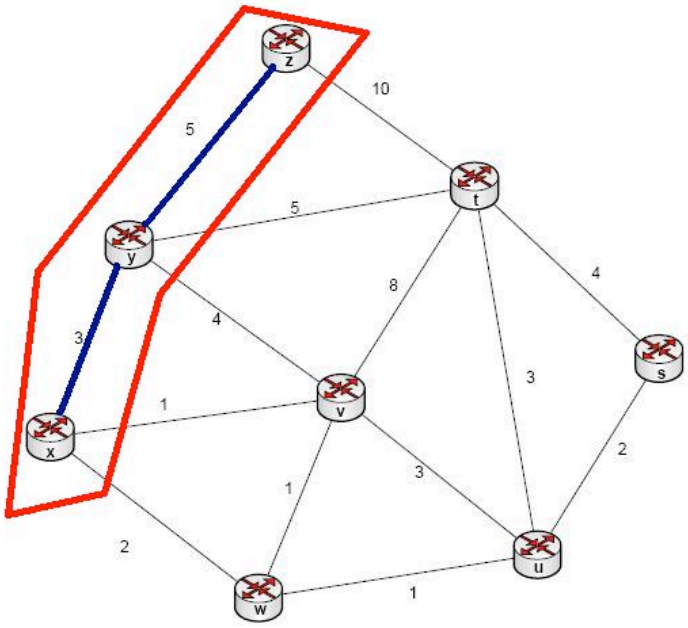
t:5

t:10

v:4

x:3

yukarıdaki listede t düğümüne iki farklı gidiş bulunmaktadır (hem z hem de y üzerinden). Biz algoritmamıza devam edip en küçük yolu bünyemize dahil edelim. En yakın komşu x:3'tür. Bu durumda üyelerimiz {z,y,x} olacak ve yollarımız {z-y:5,y-x:3} olacaktır. Bu durum aşağıdaki grafikte gösterilmiştir:



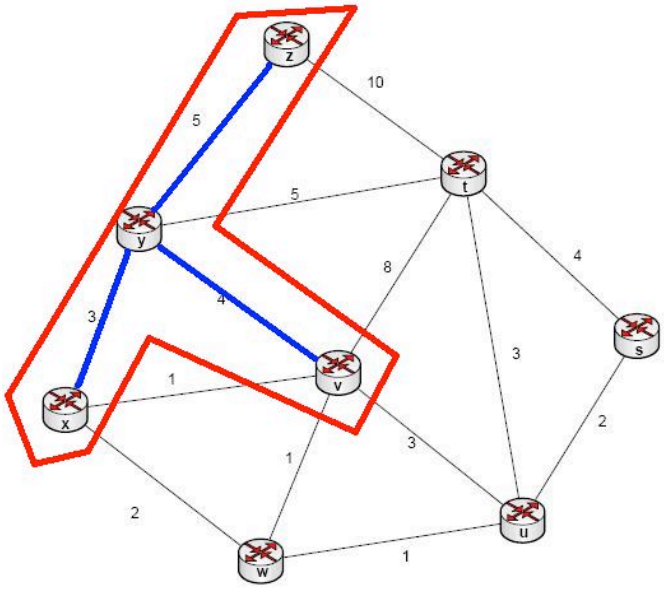
Yeniden komşularımızı listelersek:

t:5

v:1

w:2

yukarıdaki listede bünyemize aldığımız adadan, bir düğüme giden birden fazla yol bulunması durumunda en kısası alınmıştır. Bu durumda listenin en küçük elemanı olan v:1 tercih edilir ve üyelerimiz {z,y,x,v} yollarımız ise {z-y:5,y-x:3,x-v:1} olur. Durum aşağıdaki grafikte gösterilmiştir:



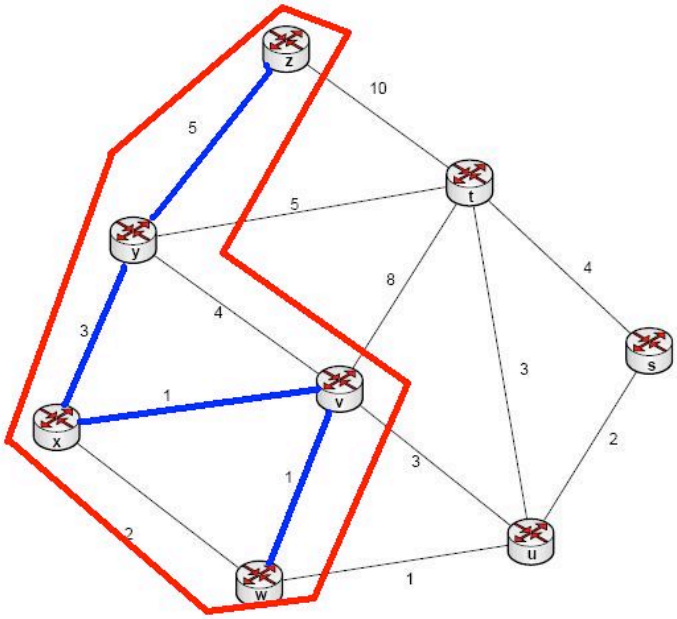
Yeniden komşularımızı listelersek:

t:5

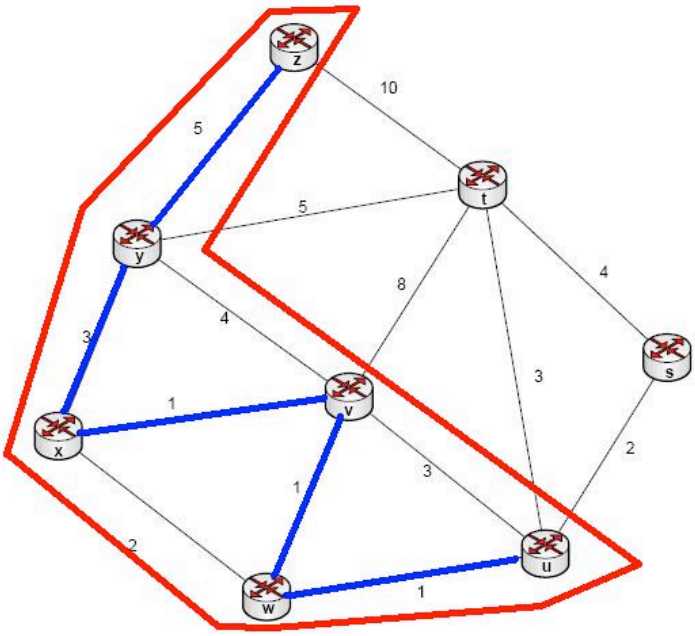
u:3

w:1

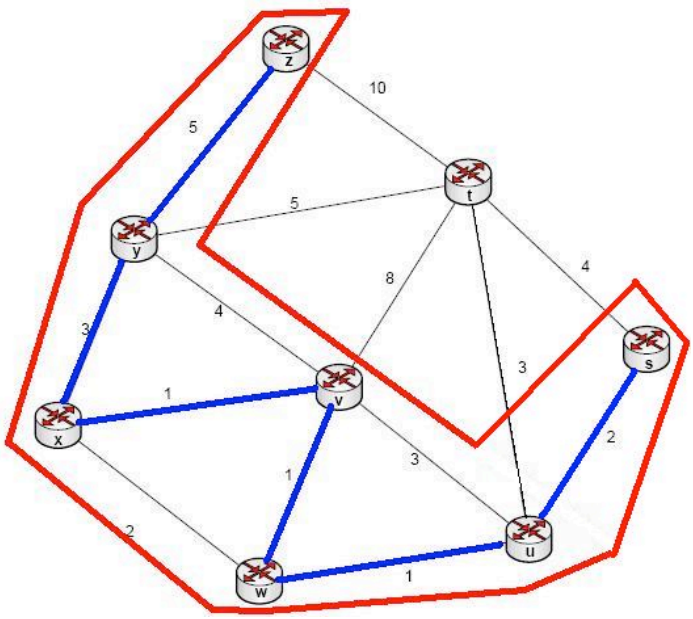
yukarıdaki listede bünyemize aldığımız adadan, bir düğüme giden birden fazla yol bulunması durumunda en kısası alınmıştır. Bu durumda listenin en küçük elemanı olan w:1 tercih edilir ve üyelerimiz {z,y,x,v,w} yollarımız ise {z-y:5,y-x:3,x-v:1,v-w:1} olur. Durum aşağıdaki grafikte gösterilmiştir:



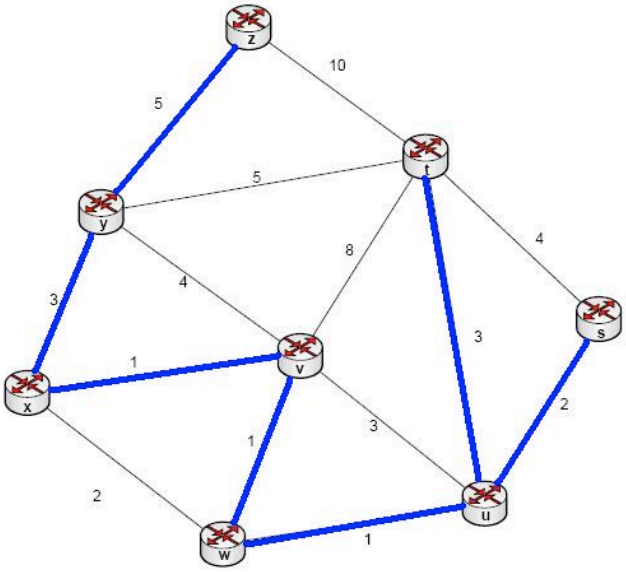
Yeniden komşularımızı listelersek:
t:5
u:1
yukarıdaki listede bünyemize aldığımız adadan, bir düğüme giden birden fazla yol bulunması durumunda en kısası alınmıştır. Bu durumda listenin en küçük elemanı olan u:1 tercih edilir ve üyelerimiz {z,y,x,v,w,u} yollarımız ise {z-y:5,y-x:3,x-v:1,v-w:1,w-u:1} olur. Durum aşağıdaki grafikte gösterilmiştir:



Yeniden komşularımızı listelersek:
t:3
s:2
yukarıdaki listede bünyemize aldığımız adadan, bir düğüme giden birden fazla yol bulunması durumunda en kısası alınmıştır. Bu durumda listenin en küçük elemanı olan s:2 tercih edilir ve üyelerimiz {z,y,x,v,w,u,s} yollarımız ise {z-y:5,y-x:3,x-v:1,v-w:1,w-u:1,u-s:2} olur. Durum aşağıdaki grafikte gösterilmiştir:



Son komşumuz olan t için en kısa ulaşım
t:3 değeridir ve u üzerinden sağlanır. Bu durumda listenin en küçük elemanı olan t:3 tercih edilir ve üyelerimiz {z,y,x,v,w,u,s,t} yollarımız ise {z-y:5,y-x:3,x-v:1,v-w:1,w-u:1,u-s:2,u-t:3} olur. Sonuçta elde edilen asgari tarama ağacı} aşağıda verilmiştir:



Kaynaklar

Arslan, M. L., & Seker, S. E. (2014). Web Based Reputation Index of Turkish Universities. International Journal of E-Education E-Business E-Management and E-Learning (IJEEEE) , 4 (3), 197-203.

Biggs, N., Lloyd, E., & Wilson, R. (1986). Graph Theory. Oxford University Press.

Chen, W.-K. (1997). Graph Theory and its Engineering Applications. World Scientific.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stei, C. (2001). Introduction to Algorithms (Vol. Second Edition). MIT Press and McGraw-Hill.

Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman.

Gibbons, A. (1985). Algorithmic Graph Theory. Cambridge University Press.

Mallows, C. L., & Sloane, N. J. (1975). Two-graphs, switching classes and Euler graphs are equal in number. SIAM Journal on Applied Mathematics , 28 (4), 876–880.

Sedgewick, R. (1983). Graph algorithms, Algorithms. Addison-Wesley.

Seker, S. E. (2015). Computerized Argument Delphi Technique. IEEE Access , 3, 368 - 380.

SEKER, S. E. (2015). Temporal logic extension for self referring, non-existence, multiple recurrence and anterior past events. Turkish Journal of Electirical Engineering and Computer Sciences , 23 (1), 212-230.

Seker, S., Altun, O., Ayan, U., & Mert, C. (2014). A Novel String Distance Function Based on Most Frequent K Characters. International Journal of Machine Learning and Computation (IJMLC) , 4 (2), 177-183.

Tutte, W. T., & Smith, C. A. (1941). On Unicursal Paths in a Network of Degree 4. American Mathematical Monthly , 48, 233–237.