GRAF TEORİSİ

İçindekiler

SORU 1: Graph ve RDF Veritabanları

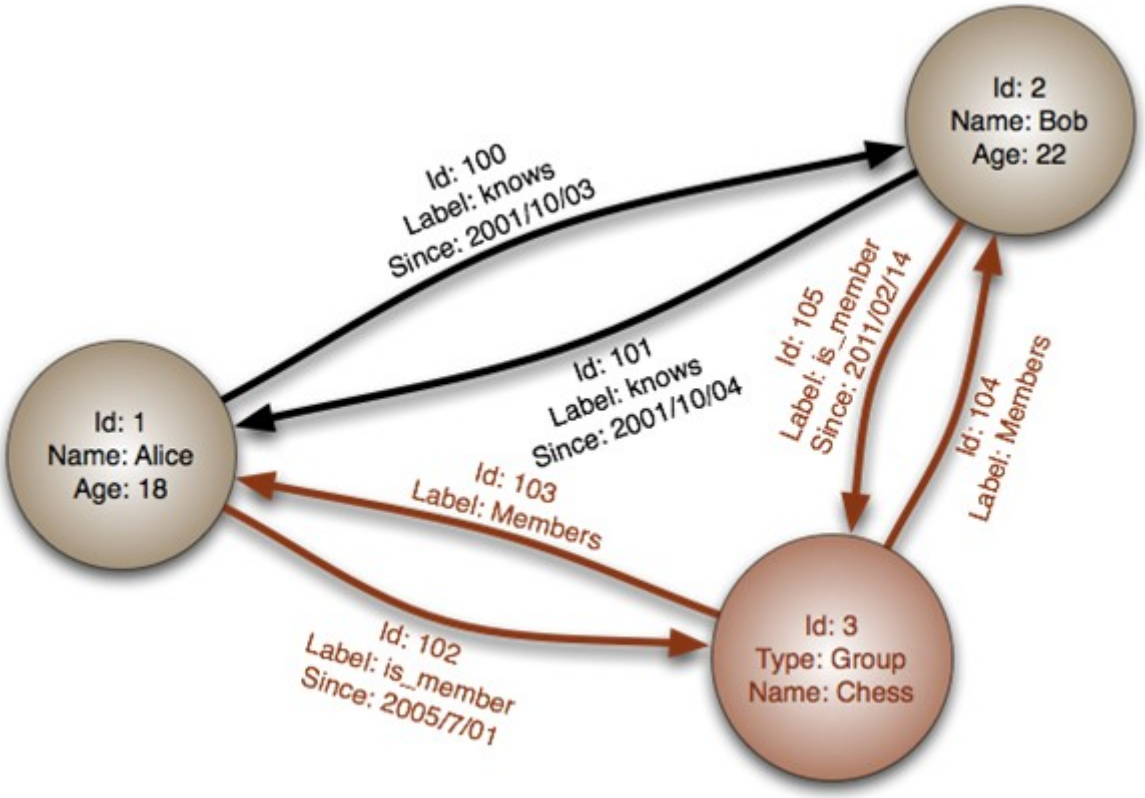
Özet

Bu doküman Graph ve RDF veritabanları hakkında genel bilgi sunmaktadır. Doküman 4 kısımda hazırlanmıştır. İlk kısımda Graph veritabanları ve tanımları açıklanmış, ikinci kısımda RDF veri tabanları açıklanmış, üçüncü kısımda Graph veritabanlarına örnekler verilmiş ve kısa açıklamalar ile resmi web siteleri sunulmuş ve son bölümde de RDF veritabanlarına örnekler verilmiş ve kısa açıklamalar ile resmi web siteleri sunulmuştur.

1. **Graph Veritabanları**

Bir graph veritabanı düğümler, kenarlar ve özelliklerle beraber graph yapılarını kullanarak veriyi sunar ve saklar. Tanımsal olarak bir graph veritabanı, indissiz yakınlık (index-free adjacency) sağlayan bir veri saklama sistemidir. Bu tanıma göre, her öğe (element) yakın olduğu öğeye doğrudan bir işaretçi (direct pointer) içerir ve indis aramaları (lookup) gereksizdir. Herhangi bir graph tutabilen genel graph veritabanları, "ağ veritabanları" ve "triplestores" gibi özelleştirilmiş graph veri tabanlarından farklıdır.

Graph veritabanları graph teorisine dayalıdır ve düğümler, özellikler ve kenarlar içerirler. Düğümler, nesneye yönelik programcıların aşina olduğu objelere çok benzemektedir.



Düğümler insan, iş, hesap gibi takip edilmek istenen varlıkları (entity) temsil eder. Özellikler, düğümlerle ilişkili olan bilgilerdir. Örneğin, eğer "wikipedia" düğümlerden biriyse, "web sitesi" veya "referans materyali" gibi özelliklere bağlanabileceği gibi düğümün hangi

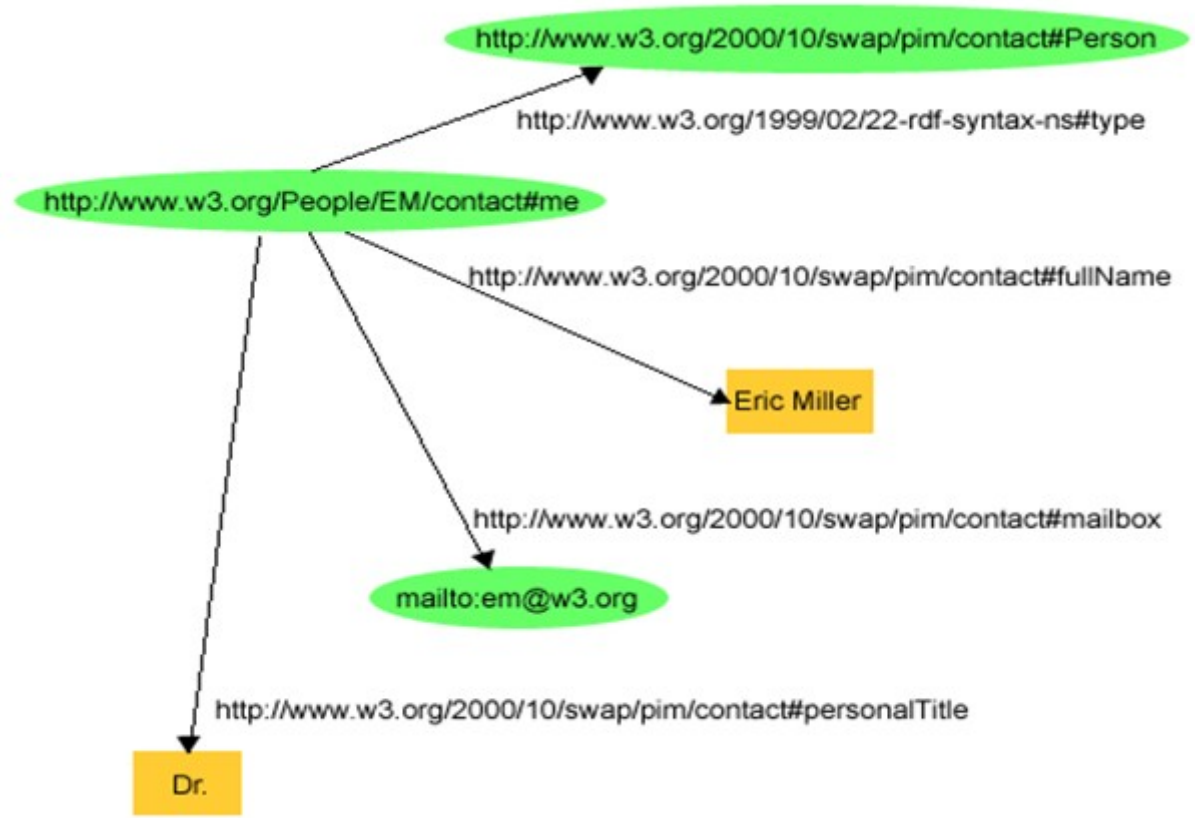
özelliklerinin veritabanı ile uyumlu olacağına bağlı olarak, "'w' ile başlayan kelimeler" gibi özelliklere de bağlanabilir.

Kenarlar düğümleri düğümlere veya düğümleri özelliklere bağlayan çizgilerdir ve iki taraf arasındaki ilişkiyi temsil eder. Önemli bilginin çoğu kenarlarda tutulur. Birisi düğümlerin, özelliklerin ve kenarların bağlantılarını incelediği zaman anlamlı örüntüler ortaya çıkar. [1]

2. RDF Veritabanları

Aslen bir "metadata" veri modeli olarak tasarlanan RDF (Resource Description Framework), World Wide Web Consortium (W3C) spesifikasyonlarının bir ailesidir. Web kaynakları içerisinde uygulanmış olan bilginin, çeşitli sözdizimi formatlarını kullanarak modellenmesi veya kavramsal olarak tanımlanması için kullanılan bir metot olagelmiştir.

RDF veri modeli, varlık-ilişki veya sınıf-diyagramları gibi klasik kavramsal modelleme yaklaşımlarına benzemektedir çünkü kaynaklar hakkında özne-yüklem-nesne ifadesi formunda değerlendirme yapma fikrine dayanmıştır. Bu ifadeler RDF terminolojisinde "Triples" olarak yer almaktadır. Özne kaynağı, yüklem kaynağın özelliklerini ifade eder ve nesne ile özne arasında bir ilişkiyi ifade eder. Örneğin, RDF'te "Gökyüzü mavi renktedir" kavramını "triple" kullanarak temsil etmenin bir yolu: "gökyüzü"nü ifade eden bir özne, "renktedir"i ifade eden bir yüklem ve "mavi"yi ifade eden bir nesnedir. Dolayısıyla RDF, nesne tabanlı tasarımdaki klasik gösterim olan varlık-özellik-değer modelinde kullanılan nesneleri, öznelerle yer değiştirir; nesne (gökyüzü), özellik (renk) ve değer (mavi). RDF birçok serileştirme biçimine (örneğin; dosya biçimi) sahip bir soyut modeldir ve dolayısıyla a bir kaynağın veya triple'ın kodlanması, biçimden biçime farklılıklar göstermektedir.



Bir RDF ifadeleri koleksiyonu özünde etiketli, yönlü (directed) bir çoklu-graph'tır. Bu yüzden RDF-tabanlı veri modeli doğal olarak belirli türdeki bilgi temsiline (knowledge representation), ilişkisel model veya diğer ontolojik modellerden daha uyumludur. Ancak pratikte RDF verisi genellikle ilişkisel veritabanı veya Triplestores (eğer her bir RDF triple'ı için bağlam da persist ediliyorsa Quadstores) adı verilen yerel temsillerle persist edilir. RDFS ve OWL'nin gösterdiği gibi, ek ontoji dilleri RDF kullanılarak inşa edilebilir. [2]

3. Graph Veritabanı Projeleri

Aşağıdaki listede iyi bilinen birçok graph veritabanı projesi listelenmiştir:

- AllegroGraph – ölçeklenebilir, yüksek performanslı RDF ve graph veritabanı
- Bigdata – yüksek derecede ölçeklenebilir RDF/Graph veritabanı, bir düğümde +10milyar kenar veya çok yüksek çıktı için kümelenmiş deployment
- CloudGraph – graph ve anahtar/değer çiftlerini veri saklamak için kullanan, disk ve hafıza tabanlı, tam işlemli (transactional) .NET graph veritabanı
- Cytoscape – açık-kaynak kodlu platform, open-source platform, biyo-enformatiğin sonucu
- DEX – Sparsity Technologies'ten yüksek performanslı bir graph veritabanı
- Filament – graph persistence çatısı ve seyirsel sorgu tarzına dayalı birleşmiş araç kutuları
- GiraffeDB – karmaşık semantikleri etkin ve erişilebilir bir yolla temsil edebilen, .NET framework 4.0 için güçlü bir grap veritabanı sistemi
- GraphBase – biçimlendirilebilir, dağıtık, yüksek performanslı ve zengin araç setli graph deposu
- HyperGraphDB – genelleştirilmiş hipergraph'ların desteklendiği, kenarların diğer kenarı işaret ettiği, açık-kaynak kodlu grap veritabanı an open-source (LGPL) graph database supporting
- InfiniteGraph – yüksek ölçeklenebilirlik dağıtık, bulut-uyumlu esnek lisanslara sahip ticari ürün
- Neo4j – açık-kaynak kodlu ve ücretli graph veritabanı
- OpenLink
  Virtuoso – yüksek performanslı RDF graph veritabanı sunucusu, yerel gömülü durum olarak deploy edilebilir, tek-durum ağ sunucusu veya çok büyük derecede ölçeklenebilir sıfır-paylaşım ağ kümesi durumu
- OrientDB – yüksek performans, açık-kaynak kodlu, doküman-graph veritabanı

4. RDF Veritabanı Projeleri

- Oracle Database Semantic Technologies – hem bedava hem de ücretli sürüm, ileri-zincirli muhakeme (forward-chaining reasoning), ölçeklenebilir
- BigData RDF Database – hem bedava hem de ücretli sürüm, yatay ölçeklenebilir, transaction destekli, yüksek G/Ç oranları, SPARQL ve RDFS
- OWLIM – hem bedava hem de ücretli sürüm, Java ile geliştirilmiş yerel RDF motoru, Sesame ve Jena, RDFS ve OWL 2RL, en iyi ölçekleme, yükleme ve değerlendirme performansı
- 4Store – bedava, ölçeklenebilir, güvenli, hızlı, güçlü
- Mulgara – açık-kaynak kodlu, ölçeklenebilir, tamamen Java
- ViziQuer – SPARQL endpoint ontolojisini taramaya ve SPARQL sorguları oluşturmaya yarayan bir araç

- SemWebCentral – açık-kaynak kodlu semantik web araçları
- Virtuoso Universal Server – RDF veri yönetimi için SPARQL uyumlu platform, SQL-RDF entegrasyonu ve RDF tabanlı bağlı-veri deployment'ı
- ROWLEX – RDF dokümanlarını kolayca yaratmak ve göstermek için kullanılan .NET kütüphanesi ve araç kutusu, RDF tripler'larının seviyesini soyutlar ve programlama işini OWL sınıflarına ve özelliklerine yükseltir.
- StrixDB: bir RDF graph deposu, SPARQL, httpd olarak kullanılabilir

Kaynakça

[1] http://en.wikipedia.org/wiki/Graph_database

[2] http://en.wikipedia.org/wiki/Resource_Description_Framework

SORU 2: Belief Propogation (İnanç Yayılımı)

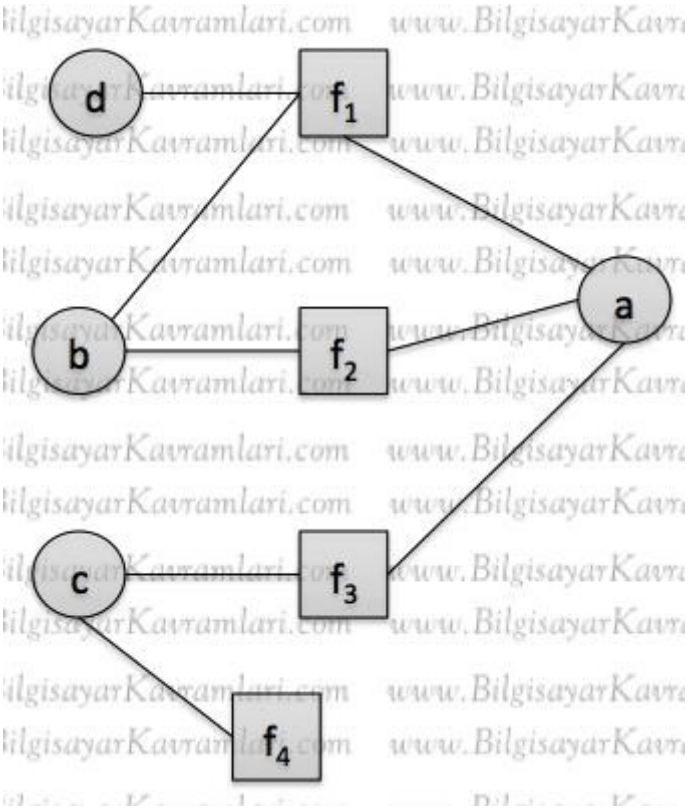
Türkçede inanç yayılması (veya iman neşri) olarak çevrilebilecek belief propagation konusu, bilgisayar bilimlerinde, makine öğrenmesi (machine learning) konusunun altında değerlendirilebilir.

Algoritma ilk olarak Judea Pearl tarafından 1982 yılında yayınlanan makalesinde duyurulmuştur. Pearl, Judea (1982). "Reverend Bayes on inference engines: A distributed hierarchical approach". *Proceedings of the Second National Conference on Artificial Intelligence*. AAAI-82: Pittsburgh, PA. Menlo Park, California: AAAI Press. pp.133–1 36.)

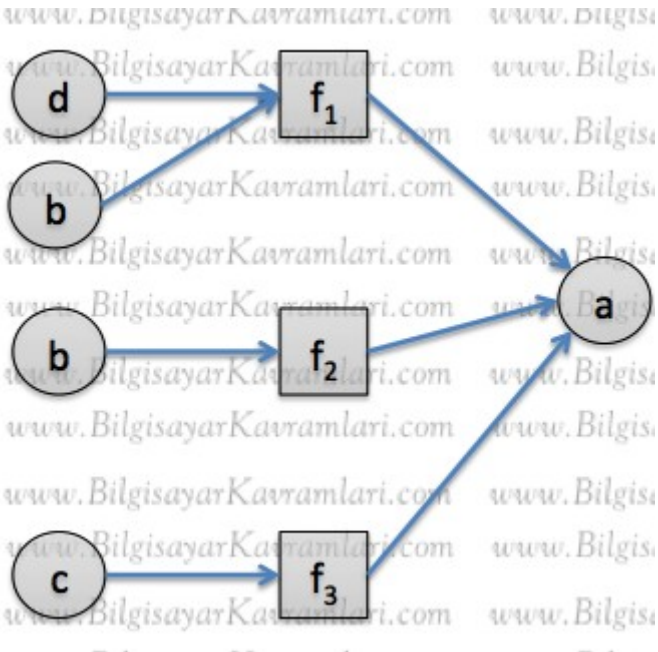
Yapı olarak mesaj geçirme (message passing) algoritmalarının bir örneği olarak görülebilir. Buradaki yayılma (neşr etmek, propagation) aslında varlıklar arasında bir mesajın geçişi anlamını taşımaktadır. Literatürde farklı kaynaklarda, toplam-çarpım mesaj geçirimi (sum-product message passing) olarak da geçmektedir. Buradaki mesaj geçişi yapılan varlıklar genelde bir şekil (Graph) üzerinde ifade edilen ve aralarında ilişkiler bulunan varlıklardır.

Esas itibariyle Markov Rastgele Alanları (markov random fields) üzerine kurulu olan inanç yayılımı konusunu bir şekil (graph) üzerindeki varlıkların birbirlerine belirli oranlarla mesaj geçirdikleri bir alan sunmaktadır.

İnanç neşriyatı için öncelikle bir problemin bir ağaç şeklinde nasıl çözülüdüğüne bakalım. Öncelikle bir çarpım şekli (factor graph) ele alıyoruz:



Bu şekilde, diyelim ki a parametresi üzerine inanç neşriyatı uygulayacağız. Bu durumda yukarıdaki şekli, aşağıdaki gibi bir ağaca (tree) dönüştürmek mümkündür:

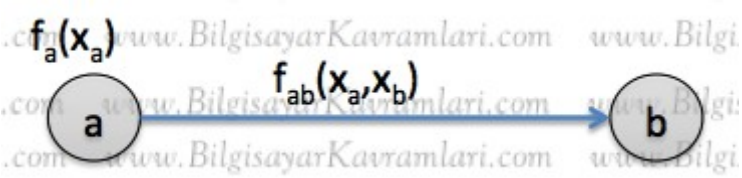


Yukarıdaki yeni şekilde, dikkat edileceği üzere, bir doğrusal ve dairesel olmayan şekil (directed acyclic graph) elde edilmiştir. Yani şekil, yönsüz şekilden (undirected graph) bir yönlü şekle (directed graph) çevrilmiştir. Bu anlamda şekli bir ağaç (tree) şeklinde ele almak mümkündür. Ayrıca şekilde b parametresi, iki ayrı fonksiyona parametre olduğu için kopyalanarak gösterilmiştir (aslında buna gerek yoktur ancak sadece konu anlaşılsın diye böyle bir yol izledim). Ayrıca sonuca etkisi olmayan f3 fonksiyonu sistemden çıkarılmıştır.

Netice olarak şekildeki 3 fonksiyonun sonuçları a parametresine birer etki olarak taşınacaktır (mesaj geçişi).

Bu anlamda, inanç neşriyatı problemini bir ağaç şeklinde düşünmek ve verilen parametreye göre problemi alt problemlere bölmek mümkündür. Hatta bu alt problemlerin birbirinden bağımsız (independent) olduğunu da söyleyebiliriz.

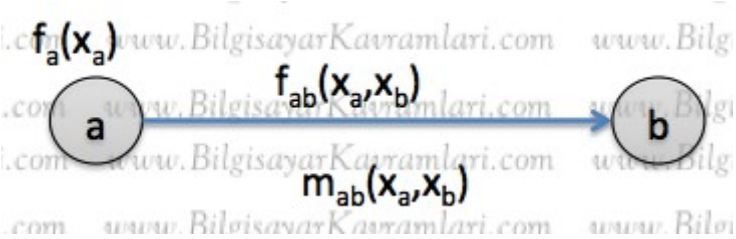
Yukarıdaki ağaç yaklaşımı üzerinden inanç neşriyatını (belief propagation) açıklamak istersek. Öncelikle iki düğüm (node) ve bir kenar (edge) üzerinde yaklaşımın nasıl çalıştığını göstermemiz gerekir:



Burada anlatılmak istenen, ağ üzerinde varlık gösteren her düğümün sonucunu döndüren bir fonksiyon ve her ilişkinin (kenar) tanımlı olduğu düğümleri aynı anda parametre olarak alan bir fonksiyonun varlığıdır. Yukarıda gösterilmemiştir ancak yine f(b) şeklinde b değerini parametre alan bir fonksiyonun varlığından da bahsedilebilir.

Aslında kenarlar üzerinde ağırlık tanımlanması halinde (ki benzer durumlar yapay sinir ağı (artificial neural network) çalışmalarında veya bayez ağlarında (bayesian network) kullanılmaktadır) bu ağarılık da bir fonksiyon olarak düşünülebilir. Örneğin f(a,b) = 0.2 gibi.

Yukardaki gösterimi biraz daha ilerletecek ve işin içerisine bir de mesaj ekleyecek olursak:

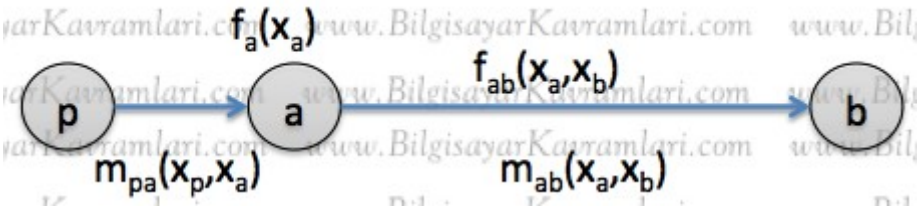


Yukarıdaki yeni şekilde eklenen m değeri, a'dan b'ye geçirilen mesajı ifade etmektedir.

Bu mesaj değeri aşağıdaki şekilde hesaplanabilir:

$$m_{ab}(x_b)=k\sum_{x_a} f_{ab}(x_a,x_b)\,f_a(x_a)\prod_{p\in N(a)/b} m_{pa}(x_a), \text{olarak yazılabilir}$$

Buradaki gösterimde kabaca tanımlı olan f fonksiyonlarından yararlanılmış ve şekildeki gösterim bir toplam-çarpım algoritması (sum-product algorithm) olarak ele alınmıştır. Hesaplamada kullanılan geçici p değeri, yukarıdaki a düğümüne etki eden herhangi bir p hesaplamasını ifade etmektedir. Yani a düğümüne kadar gelen etkiler çarpım sembolü ile hesaplanmıştır. Bu durum aşağıdaki şekilde düşünülebilir:



Yani denklemin çarpım sembolü kısmında bu p düğümünün etkisi ile a'ya bağlanan kenardan gelen fonksiyon değerinin a üzerindeki etkisi alınmıştır. Bu şekilde, a'ya etki eden bütün düğüm ve kenar bağlantıları (ki a'ya bağlı sadece p gösterilmiştir ancak başkaları da olabilir) a üzerinde çarpım sembolü ile ifade edildikten sonra geriye a gibi b üzerinde etkisi bulunan bütün düğümlerin toplamlarını hesaplamak kalmıştır. Denklemin toplam sembolü ile gösterilen kısmı da bu toplamayı yapmaktadır.

Yukarıdaki yaklaşımı toplamdan çıkarıp azami değeri alacak şekle getirirsek:

$$m_{ab}(x_b)=k \max_{x_{ab}} f(x_a,x_b) f_a(x_a) \prod_{p\in N(a)/b} m_{pa}(x_a), \text{olarak yazılabilir}$$

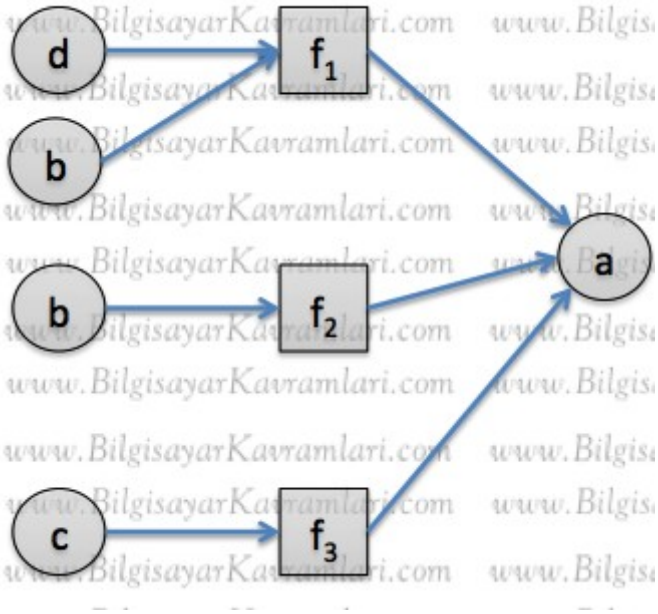
Yeni yaklaşımımızda azami değerler alınarak parametrelerin arasında en büyük değeri yani en aykırı değeri (marginal) bulmak amaçlanmıştır. Bu değerlere aykırı değer veya azami-aykırı değer (marginal veya max-marginal) ismi verilir. Kısaca aşağıdaki şekilde de gösterilebilir:

$$p_a(x_a)=k f_a(x_a) \prod_{p\in N(a)} m_{pa}(x_a), \text{olarak yazılabilir}$$

Bu anlamda, önce çarpım sembolünü işletmek (iterate) ve bütün değişkenler için bir bir çalıştırmak mümkündür. Her çalışan değişken için aslında sistemden bir değişkenin kaldırıldığını söyleyebiliriz.

Paralel Mesaj Geçirimi

Bu parametre kaldırma işlemini paralel hale getirmek de mümkündür. Yazının başında verilen şekli hatırlayalım:



Bu şekilde bulunan f2 ve f3 fonksiyonları veya f1 ve f3 fonksiyonları, birbirinden bağımsız olarak hesaplanabilir. F1 ve f2 fonksiyonları ise aynı değişkene (b) bağlı oldukları için bazı durumlarda birbirini beklemek zorundadır.

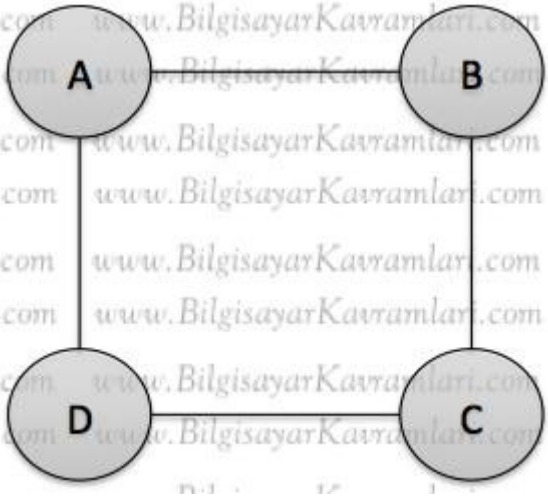
Bu paralel işleme işine de paralel mesaj geçirme (parallel message passing) ismi verilir.

Döngüsel İnanç Neşriyatı (Loopy Belief Propagation)

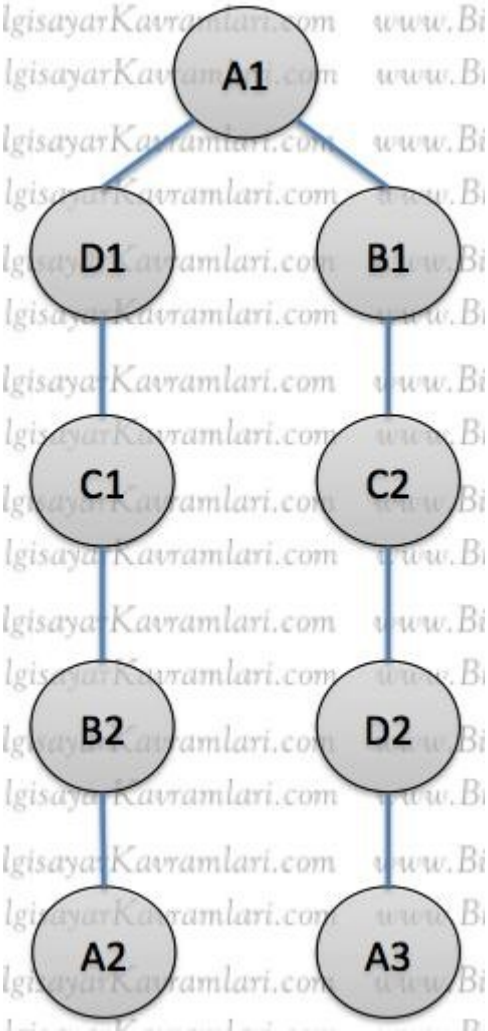
Gelelim döngüsel inanç neşriyatına (döngüsel inanç yayılımı). Bu yaklaşımda, şekilde bir dairesel (cyclic) özellik olacağı kabulü bulunur. Ancak döngüsel olarak yapılan yaklaşımın her zaman bir noktada toplanması/birleşmesi garanti edilemez.

Hatta bir noktada toplanması mümkün olsa bile, doğru marjinal değerlerde toplanması garanti edilemez.

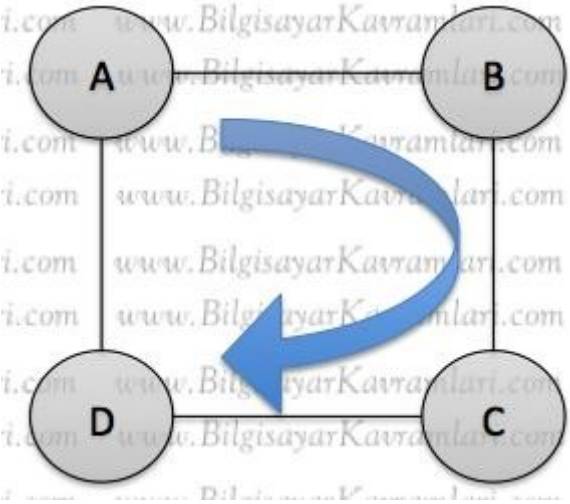
Örneğin tek döngü içeren aşağıdaki şekli ele alalım:



Bu markof rastgele alanından aşağıdaki sargısız şekli (unwrapped graph) elde etmek mümkündür (okuyucu detayları için http://www.bilgisayarkavramlari.com/2012/05/05/unwrapped-graphs-sargisiz-sekiller/ bağlantısındaki yazıya başvurabilir):



Dolayısıyla yukarıdaki yaklaşım kullanılarak mesaj güncellemesi iki farklı yönde işletilebilir. Buradaki amaç inanç ağının, döngüsel bir markof alanında çalıştırılmasıdır. Çalışmanın başarılı olması için yukarıda gösterildiği gibi sargısız şekil (unwrapped graph) elde edilip üzerinde çalışılabileceği gibi, orjinal markof rastgele alanında da bir yön belirlenerek çalışılınabilir:



Örneğin yukarıdaki ağaç şekili çıkarılırken, A düğümünden başlanarak saat yönünde dönülen bir kol (ağacın sağ kolu) ve bu yönün tam tersi istikamette dönülen diğer bir kol (ağacın sol kolu) çizilmiştir.

Yukarıdaki şekilde bir döngü içeren inanç ağı, kararlı hale (steady state) ulaşana kadar parametre güncellemesi ile işlenir. Yani ağda yayılım (neşriyat, propagation) sürekli devam eder. Neticede kararlı bir hal alır ve durulur.

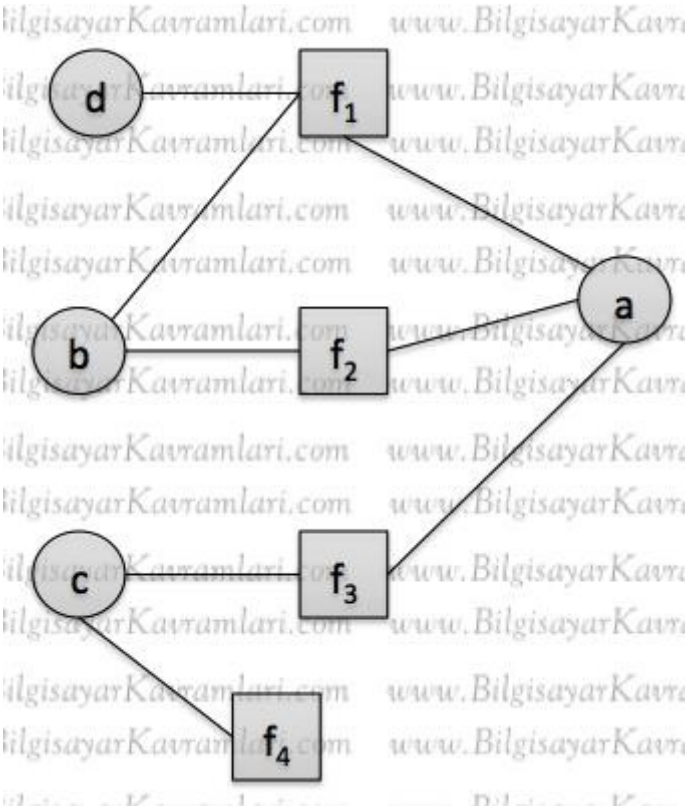
SORU 3: Sum Product Algorithms (Toplam Çarpım Algoritmaları)

Toplam çarpım algoritmaları (sum-product algorithms), çeşitli istatistiksel ve hesaplamalı çalışmalarda, birden fazla varlığın ürettiği verilerin işlenmesi için kullanılır. Buradaki amaç, birbiri üzerinde etkisi bulunan bayez ağı (bayesian network) veya markof rastgele alanı (markov random field) gibi yapıları modellemek ve mesaj geçirmek (message passing) aracılığı ile çözümlemektir. Yapısal olarak bir şekil (graph) üzerinde çalışan bu algoritmaların üzerinde çalıştığı şekillere çarpan şekli (factor graph) ismi de verilmektedir.

Konuyu anlamak için, aşağıdaki fonksiyonu ele alalım:

$$f(a,b,c,d)=f_1(a,b,d)\,f_2(a,b)\,f_3(a,c)\,f_4(c)$$

Bu fonksiyonun çarpan şekli aşağıda verilmiştir:



Şekilde görüldüğü üzere fonksiyonun parametreleri ve fonksiyonu oluşturan alt fonksiyon çarpımları arasındaki ilişkiler modellenmiştir.

Öncelikle mesaj geçirme işlemini anlamamız gerekir. Bunun için yukarıdaki fonksiyon üzerine yani f(a,b,c,d) fonksiyonu üzerine toplam çarpım yöntemini uyguluyoruz. Burada kullanacağımız ilk formül:

$$m_{a\to i}=\sum_{x_a/x_i} f_a(x_a) \prod_{j\in N(a)/i} n_{j\to a}(x_j)$$ *formülüdür.*

Kısaca anlatacak olursak, her i. elemena mesaj geçirme işlemi için i. elemanı dışlar şekilde bütün fonksiyonların toplamıdır. Yani f(a,b,c,d) fonksiyonunu oluşturan çarpanların hepsi için ayrı ayrı parametrelerin etkileri toplanır.

Yukarıdaki denklemde bulunan çarpım kısmını da açmakta yarar var:

$$n_{i\to a}(x_i)=\prod_{b\in N(i)/a} m_{b\to i}(x_i)$$

olarak yazılan kısımda, her parametre için, o parametrenin ilgili olduğu fonksiyonların çarpımı alınır. Diğer bir deyişle örneğin f(a,b,c) fonksiyonunda b parametresi ile ilgileniyorsak, b parametresi bulunmayan çarpan fonksiyonları ile ilgilenmeyiz.

Şimdi yukarıdaki gösterimler ışığında f(a,b,c) fonksiyonu üzerinde mesaj geçirimini göstermeye çalışalım. Örneğin $f_a(x_a)$ gösteriminin açık halini yazarak başlayalım:

$$f_a(x_a) = f_1(a,b,d) f_2(a,b) f_3(a,c)$$

olacaktır ve $f_4(c)$ fonksiyonu bu denklemde bulunmayacaktır çünkü a parametresini içermemektedir. Benzer şekilde diğer fonksiyonları yazacak olursak:

$$f_b(x_b) = f_1(a,b,d) f_2(a,b)$$

$$f_c(x_c) = f_3(a,c) f_4(c)$$

$$f_d(x_d) = f_1(a,b,d)$$

olarak yazılabilir. Şimdi bu fonksiyon çarpanlarının toplam değerlerini alacak olursak:

$$f_1(x_a) = \sum_a \left(\sum_c f_3(a,c) f_4(c) \right) \left(\sum_b f_2(a,b) \left(\sum_d f_1(a,b,d) \right) \right)$$

yukarıdaki şekile yazılabilir. Burada dikkat edilirse, d geçen terimler (ki sadece f1′dir) toplandıktan sonra bu fonksiyon içerisinde b de geçtiği için b toplamına dahil edilmiştir. Benzer şekilde içerisinde a geçtiği için c,b ve d toplamlarının tamamı, a toplamına dahil edilirken, c toplamına b ve d toplamları alınmamıştır. Bunun sebebi b ve d toplamlarında bulunan fonksiyonlarda c değeri geçmiyor olmasıdır.

Buradaki algoritma aşağıdaki şekilde yazılabilir:

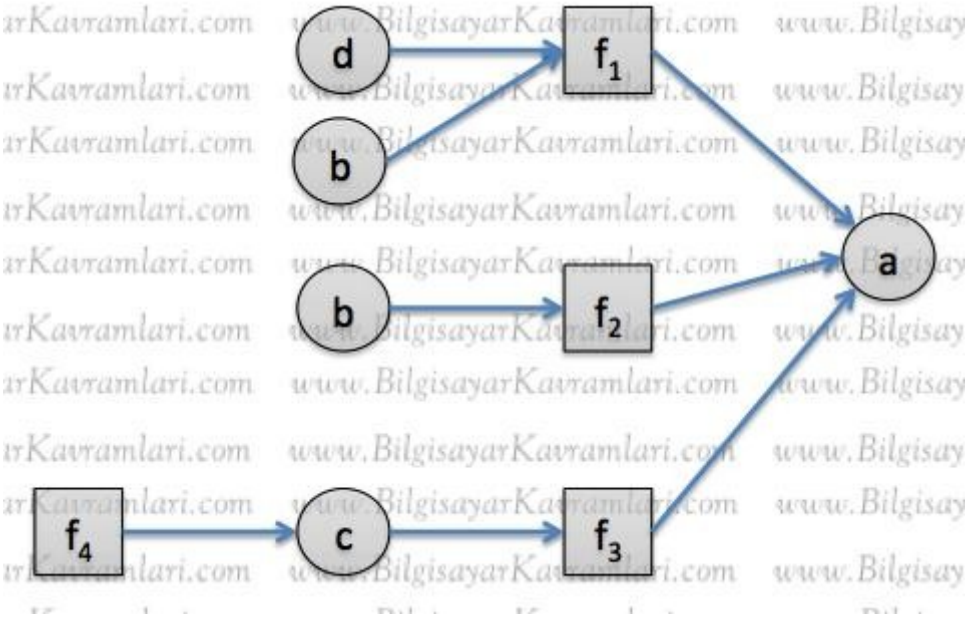
Çarpma Kuralı:

Bir değişken düğümünde, çocukların çarpımını al

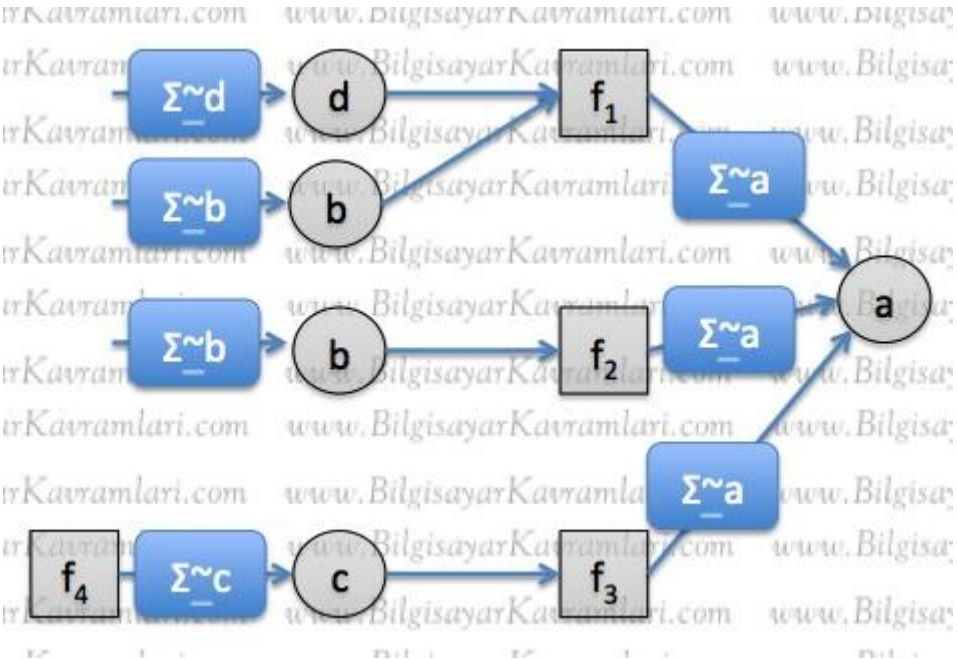
Toplama Kuralı:

Bir fonksiyon düğümünde, f fonksiyonunun çocuklarının çarpımını al ve f'in atası üzerinde eksik-toplam kuralını uygula.

Bu modellemede bir mesaj geçirme (message passing) işlemi yapılmak istenirse, ve örneğin a parametresinde mesaj toplanacak olursa, aşağıdaki şekilde bir ağaç elde etmek mümkündür:

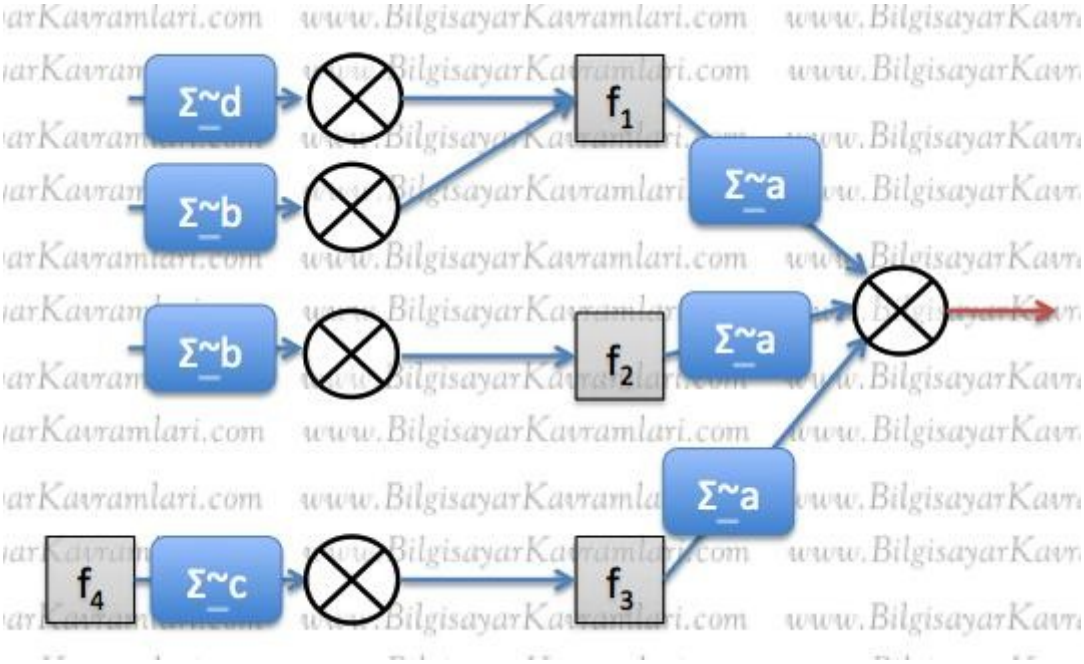


Yukarıdaki bu ağaca toplam-çarpım algoritmasını uygulayacak olursak durum aşağıdaki şekilde gösterilebilir (toplam kuralı):



Öncelikle her parametreye taşınan eksik toplamlar hesaplanır. Yukarıdaki şekilde b ve d parametreleri için de benzer bir taşıma yapılmıştır. Ancak şayet bu değerler yoksa, yani b ve d parametrelerini etkileyen başka fonksiyonlar yoksa bu değerler alınmaz. Yukarıdaki f(a,b,c) fonksiyonumuz için böyle bir durum söz konusu değildir dolayısıyla bu değerler yok hükmündedir ancak konunun anlaşılması açısından bu değerler şekilde gösterilmiştir.

Ardından her parametre için ilgili fonksiyonun çarpım değerleri hesaplanır (çarpım kuralı):

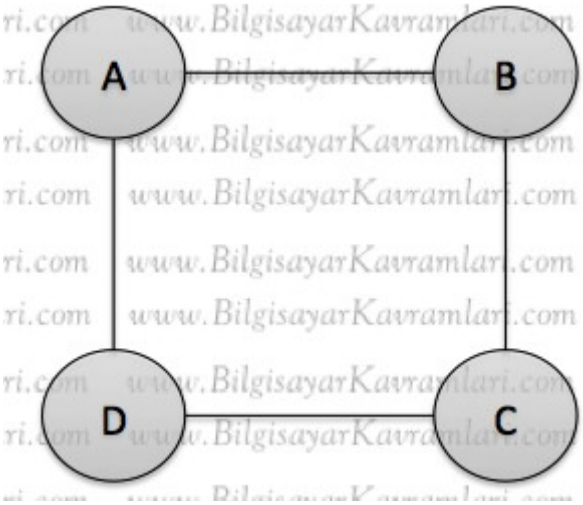


Sonuçta elde edilen değer kırmızı okla gösterilen çıkış değeridir. Bu değer a parametresi için hesaplanmışıtr. Hesaplanan parametre değişmesi halinde çıkış bu parametreden alınacaktır.

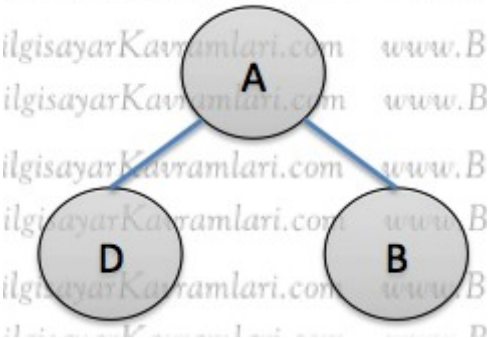
SORU 4: unwrapped graphs (sargısız şekiller)

Bu yazının amacı, şekil teorisinde (graph theory) kullanılan sargısız ağaç (unwrapped tree) kavramını açıklamaktır. Şekil teorisi üzerine kurulu pek çok çalışmada sıkça geçmekte olan bu kavram, basitçe şeklin ifade ettiği değerlerin sadeleştirilmesi ve şekildeki döngülerin açılarak şeklin bir ağaca dönüştürülmesinden ibarettir.

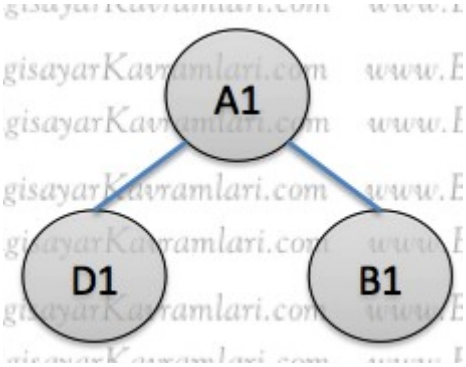
Konuyu basit bir döngüyü (cycle) açarak anlamaya başlayalım:



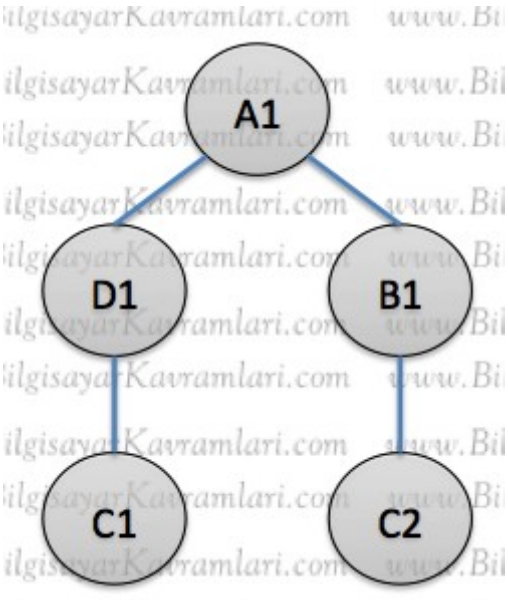
Yukarıdaki şekli sargısız hale getirmek için öncelikle bir düğüm seçilerek işe başlanır. Biz örneğimizde A düğümünü (node) kök düğüm (root) olarak seçerek başlayalım ve bu düğümün komşuluk ilişkisini bir ağaç şeklinde gösterelim:



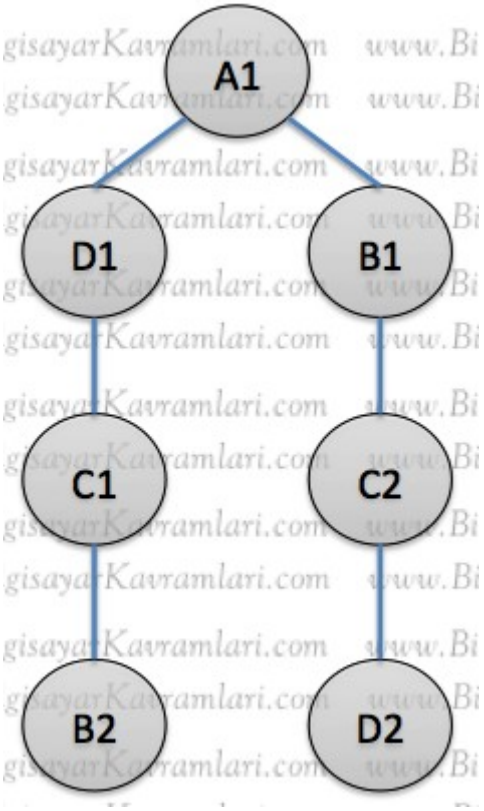
Şekilde görüldüğü üzere A düğümü ve bu düğümün komşuları ile olan ilişkisi modellenmiştir. Bu ilişkide daha sonra tekrar edecek düğümleri karıştırmamak için düğümlere isimlerinin yanında bir de sayılar ekleyelim:



Şimdi yukarıda, komşuluk ilişkileri henüz işaretlenmemiş olan D1 ve B1 düğümlerinin de komşularını işaretleyelim. Bu işaretleme sırasında dikkat edeceğimiz bir husus, orjinal şeklimizde olan ve D-A ve B-A bağlantılarının geldiğimiz yön itibariyle şeklin son halinde ifade ediliyor olduğudur. Yani zaten D ve B düğümlerine, D-A ve B-A kenarlarını dolaşarak geldik. Bu gelinen kenarlardan bir kere daha geçmiyoruz. Dolayısıyla bu ilişkiler ikinci kere gösterilmez. Bunun yerine henüz gösterilmemiş olan D-C ve B-C ilişkilerini gösteriyoruz.



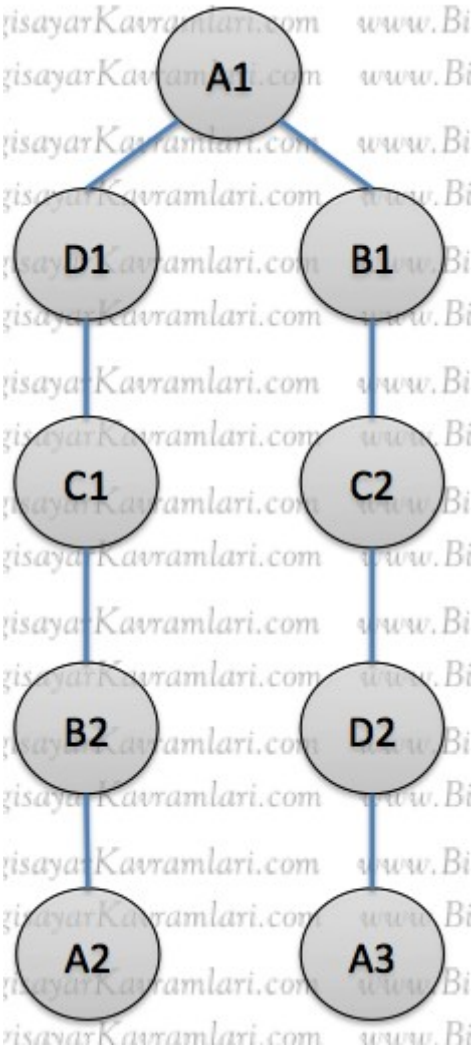
Şekilde iki farklı C değeri bulunduğu için ikisini de farklı sayılar ile ifade ettik. Zaten bu noktada dikkat edilirse bir şeklin nasıl sargısız hale getirildiği anlaşılır. Yani her farklı komşuluk düğüm kopyalaması ile farklı bir ilişki olarak ifade edilmiştir. Aynı yaklaşımla devam edersek, aşağıdaki şekli elde ederiz:



Şeklin bu son halinde iki yolu sırasıyla yazacak olursak,

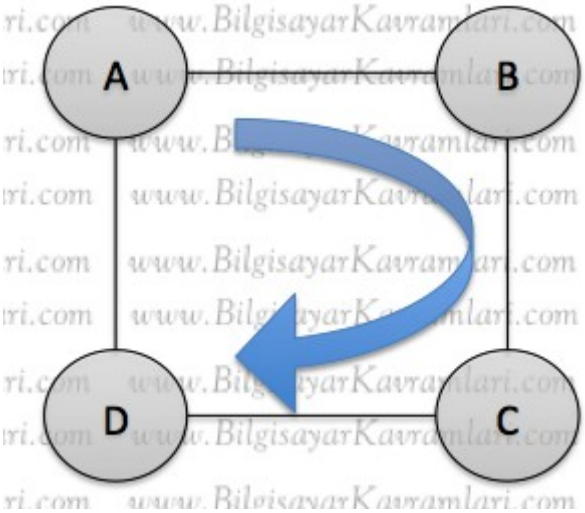
A-D-C-B yolu solda

A-B-C-D yolu ise sağda görülmektedir. Devam edelim:



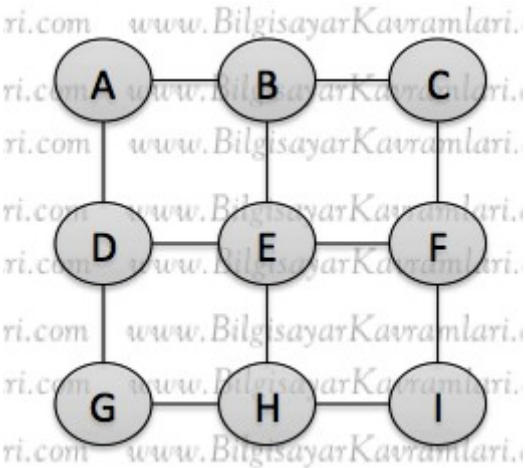
Son halinde şekil yeniden A düğümüne ulaşmıştır. Bu noktada bazı kaynaklar durarak bu sargısız hale getirmenin yeterli olduğunu savunur ve bu açılmış şekil üzerinden işlem yapar. Bazı kaynaklar ise sonsuza kadar giden bir zincir (yol, path, chain) oluştuğunu kabul eder.

Kısacası, şekil, başlangıç düğümüne gelinene kadar iki farklı yönde dönülüyor:

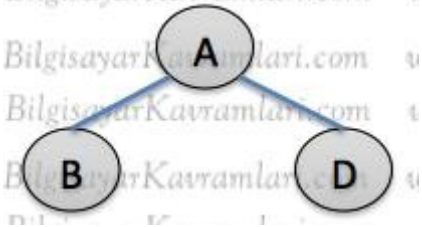


Yukarıdaki şekilde görülen saat yönündeki dönme neticesinde, sargısız düğümün sağ kolu ve bu yönün tam tersi istikametteki dönme neticesinde de sargısız şeklin sol kolundaki yol (path) ortaya çıkmaktadır.

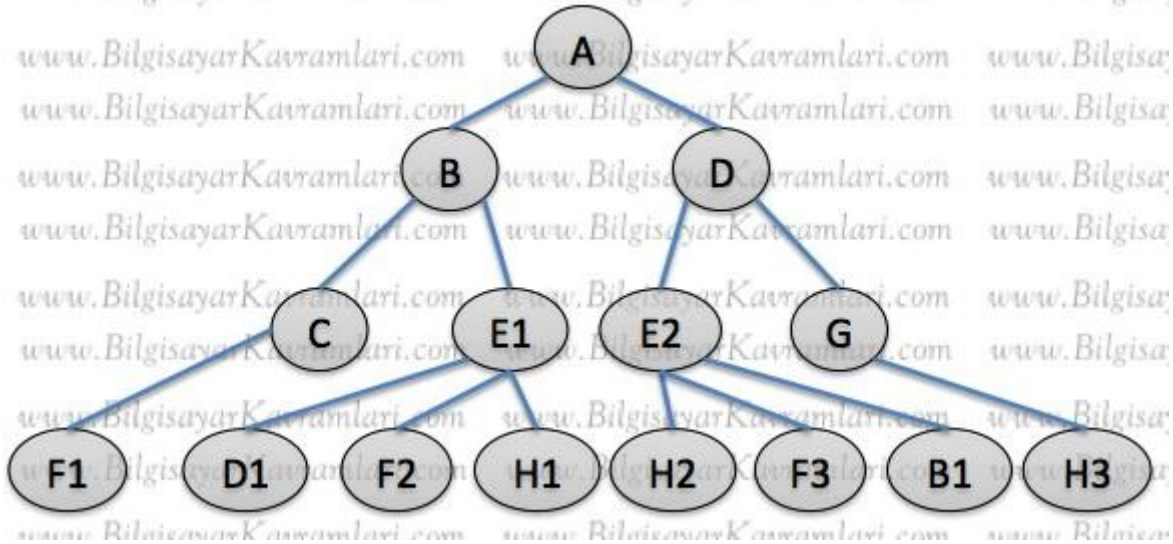
Yukarıdaki açılımı basit bir döngü (single cycle) için yaptık ancak birden fazla döngü bulunması hali de incelemeye değer. Örneğin aşağıdaki şekli ele alalım:



Bu yeni şekilde, tahmin edileceği üzere oldukça büyük bir ağaç çıkacaktır. Bu yüzden bir iki seviye ilerleyerek ağacın nasıl açılıdğını gösterip bırakacağım. Başlangıç düğümü olarak (kök düğüm, root) A seçiyorum. Bu düğümün farklı şekillerde seçilebileceğini hatırlayınız:



Şekli ilerletirsek, aşağıdaki gibi bir durum ile devam ederiz:

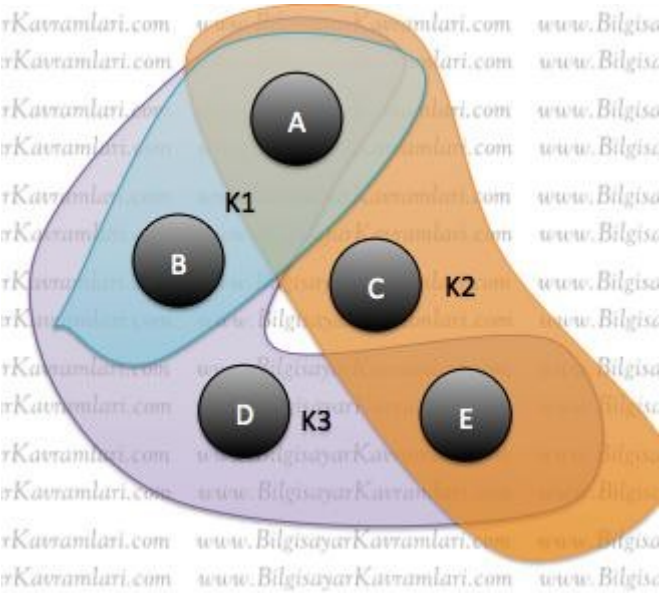


Elbette şeklin, bundan sonra da ilerletilmesi mümkündür ancak şekil bir iki seviyede çok hızlı bir şekilde genişlemektedir, konunun anlaşıldığını düşünerek bu seviyede kesiyorum.

SORU 5: HyperGraph (HiperGraf, İleri Şekil)

Bu yazının amacı, hipergraf (ileri şekil, hypergraph) konusunu anlatmaktır. Matematiksel bir terim olan hipergraf kavramı, bilgisayar bilimlerinin çeşitli alanlarında kullanılmaktadır.

Tanım itibariyle bir kenarın (edge) çok sayıdaki düğüme (node) bağlanabildiği özel şekillerdir. Yani, normalde bir şekilde (graph) bir kenar (edge), iki düğüm (node) arasında tanımlıyken ve bu iki düğümü birleştiriyorken, hipergraflarda istenilen sayıdaki düğümü birleştirebilir. Burada istenilen sayı ile kast edilen normal şekillerde olduğu gibi iki düğüm olabilirken tek bir düğüm de olabilir. Normal şekillerde bir kenar tek bir düğüme bağlı olamaz, mutlaka iki düğüme bağlı olmalıdır ama hipergraflarda tek bir düğüme, üç ayrı düğüme veya yirmi ayrı düğüme bağlı olabilir.



Yukarıdaki şekilde, 5 düğüm (node) ve 3 kenar (edge) verilmiştir. Bu şekildeki kenar kümeleri aşağıdaki şekildedir:

K1: { A, B}

K2: { A, C, E}

K3: { A, B, D, E}

Görüldüğü üzere, bir kenar klasik graftan farklı olarak çok sayıda düğüm içerebilmektedir. Bu durumda M adet düğüm ve I adet kenar içeren bir hipergrafın tanımı aşağıdaki şekilde yapılabilir:

H (D , K)

Buradaki X, düğümler kümesi ve K ise kenarlar kümesidir. Ancak belirtmek gerekir ki K, aslında bir kümeler kümesidir. Yani K'nın her elemanı D'nin alt kümesinden oluşan bir kümedir.

Normal bir şekilde (graph)

G (D, K)

şeklinde yapılan tanımda, K kümesi ikili elemanlardan oluşan bir kümeyken, bir hiper graf için eleman sayısı asgari 1 ve azami M olan kümelerden oluşan bir kümedir.

Bu durum aşağıdaki şekilde tanımlanabilir:

H (D , K) için

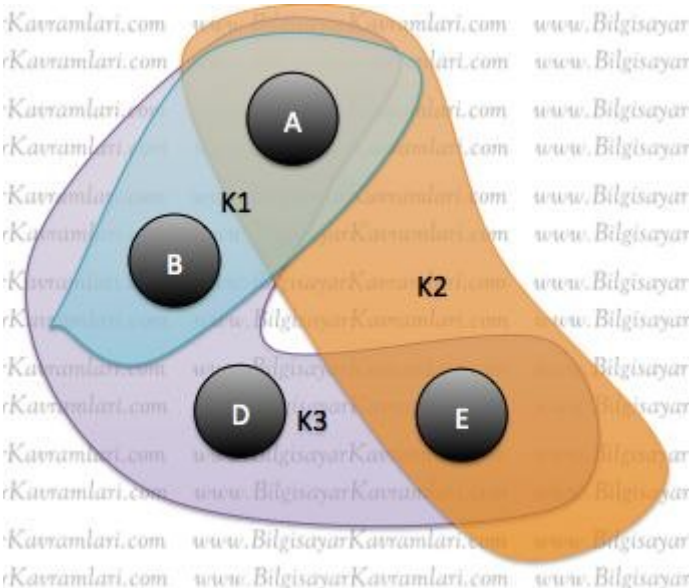
$$D=\{d_m|m\in M\}\ ve\ K=\{k_i|i\in I\ ,k_i\subseteq D\}\ olur$$

Yani diğer bir deyişle, D kümesindeki elemanlar M düğüm için 1′den M'e kadar dm ile gösterilirken, K kümesindeki elemanlar 1′den I'ye kadar ki ile gösterilmektedir. Ayrıca hiper grafın en önemli özelliği olan ki'lerden oluşan küme, D kümesinin herhangi bir alt kümesi olabilir.

Klasik şekillerde (graph) alt şekil (subgraph) tanımı yapılabildiği gibi, hiper graflar için de benzer tanımlamalar yapmak mümkündür. Ancak ancak, hipergraflarda, kenarların belirlediği sayısallık (cardinality) değerleri farklılık gösterdiği için birden fazla alt şekil (subgrah) tanımına ihtiyaç duyulmuştur. Bu tanımlar aşağıda, başlıklar halinde sunulmuştur.

Alt HiperGraf (Sub HyperGraph):

Ayrıca literatürde geçen alt hiper graf (sub-hypergraph) kavramı, bir hiper grafın düğümlerinin bir kısmının içerilmediği hiper graftır. Örneğin yukarıda verilen hiper grafa H1 diyecek olursak ve bu hipergraftan C düğümünü çıkardığımız yeni hiper grafa H2 diyecek olursak. H2 hipergrafı, H1 hipergrafının bir alt hiper grafıdır denilebilir:



Şekilde görülen yeni hiper graf için kenar kümelerinin yeni tanımı aşağıdaki şekilde olacaktır:

K1: { A, B}

K2: { A, E}

K3: { A, B, D, E}

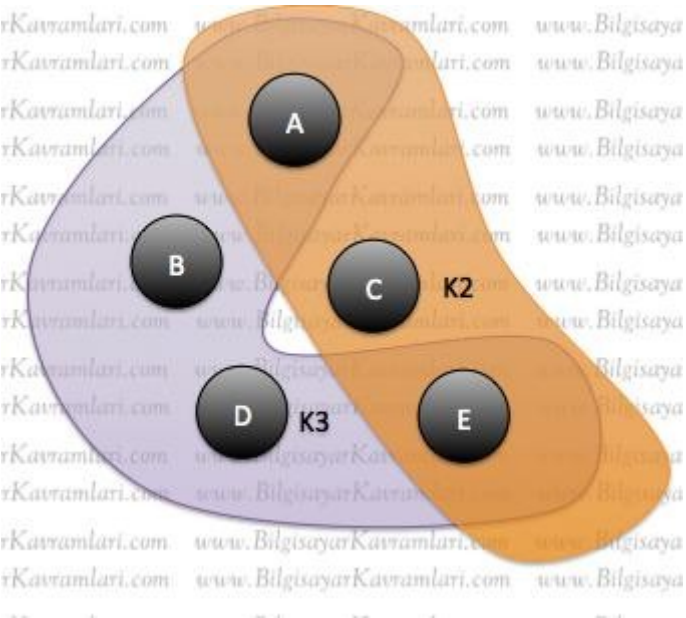
Bu yeni hipergrafta, normal bir hipergrafın taşıdığı bütün özellikler bulunmaktadır. Buna göre bir althipergraf aşağıdaki şekilde tanımlanabilir:

$$H_A = (A, \{k_i \cap A | k_i \cap A \neq \emptyset\}) olur.$$

Yukarıdaki tanımda, normal bir hipergrafın düğümlerini gösteren D kümesinin bir alt kümesi olan A kümesi tanımlanmıştır ($A \subset D$) ve buna göre kenarlar kümesi bu A alt kümesinde bulunan düğümleri bir şekilde ilgilendiren alt küme olmaktadır. Öyleki, şayet bu kenarlar alt kümesi DA için boş kümeye dönüşüyorsa, bu küme alınmaz. Yani D kümesinden çıkarılan düğümlerden sonra herhangi bir kenar kümesi boş küme oluyorsa, bu kenar alt hipergrafta bulundurulmaz.

Kısmi HiperGraf (partial hyper graph):

Bir hiper grafın bazı kenarlarının olmadığı hiper graflara verilen isimdir.



Örneğin yukarıdaki yeni hipergrafta, K1 kenarı bulunmamaktadır. Bununla birlikte bütün düğümler (nodes) tam olarak yer almaktadır. Yukarıdaki hiper grafa H3 ismini verecek olursak, H3, H1′in bir kısmi hipergrafıdır denilebilir. Bu hiper grafta bulunan kenar kümeleri aşağıdaki şekilde tanımlanabilir:

K2 = {A, C , E }

K3 = {A, B , D , E}

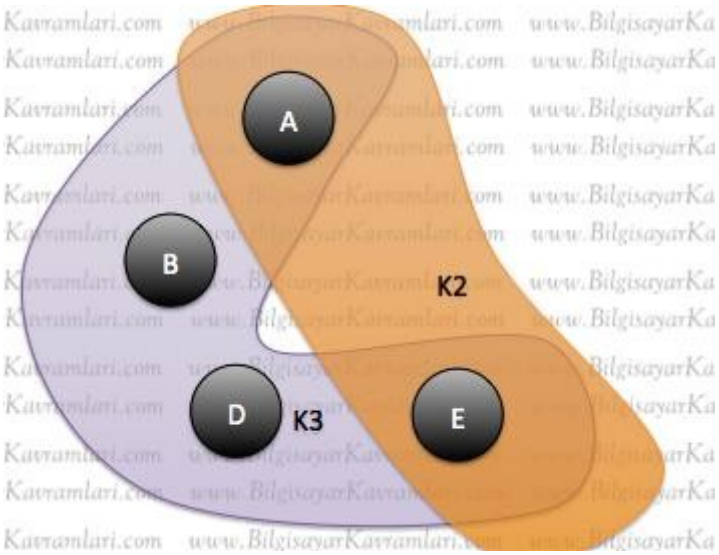
Bu tanıma göre, şayet $J \subset K$ tanımı yapılırsa,

$$H_{kısmi} = (D, \{k_i \in J\})$$

şeklinde bir kısmi hiper graf tanımlanabilir.

Bölgesel Hiper Graf (Section Hyper Graph):

Bu kavram, hem kısmi hem de alt hipergraf kavramlarının birleşimidir. Yani hem kenar hem de düğümler, Orijinal hiper grafın alt kümesi olarak kabul edilebilir.



Yukarıdaki şekilde, hem bir düğüm eksik (C düğümü) hem de bir kenar eksiktir (K1 kenarı) bu durumda bu hiper graf, Orijinal hiper grafın ne kısmi ne de alt hiper grafıdır. Bunun yerine bölgesel hipergrafıdır terimi kullanılır.

İki parçalı graflar (bipartite graph)

Herhangi bir H hipergrafı, iki parçalı graf (bi-partite graph) olarak gösterilebilir. Bunun için D ve K kümelerinin iki parçalı graftaki bölümleri ifade etmesi gerekir.

Herhangi bir düğüm d için, sadece tek bir kenar k tarafından içerilmelidir.

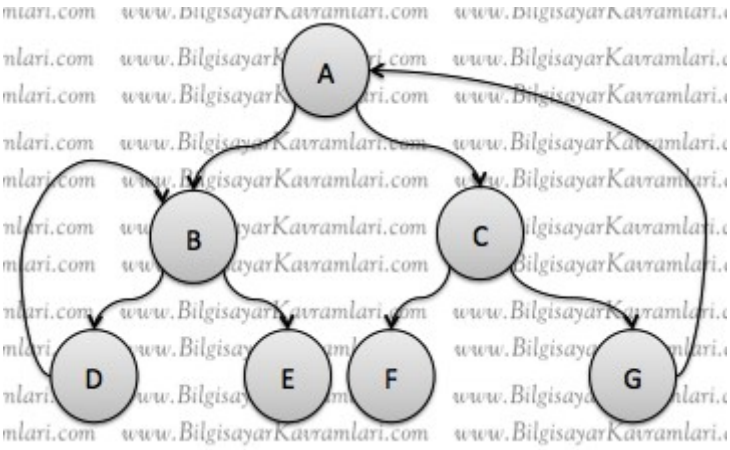
SORU 6: Yinelemeli Derinlik Araması (Iterative Deepining Search)

Bilgisayar bilimlerinin çeşitli alanlarında (örneğin yapay zeka, veri yapıları veya şekil kuramı (graph theory) gibi) kullanılan arama algoritmalarından birsidir.

Algoritma, derin öncelikli drama (depth first search) üzerine kurulu olduğu için, literatürde "iterative deepening depth first search (yinelemeli derinleşen, derin öncelikli arama)" olarak da geçmektedir.

Algoritma basitçe derinlik değerini bir değişkende tutmakta ve bu değeri her adımda arttırmaktadır.
Yineleme yapısı (iteration) basit bir döngü (loop) olarak düşünülebilir ve her adımda derinliğin, bir döngü değişkeni (loop variable) gibi düşünülerek derinleştiği kabul edilebilir.

Örneğin aşağıdaki şekli (graph) ele alalım:



Ağaçta görüldüğü üzere iki adet döngü (cycle) bulunmaktadır. Iterative Deepening search algoritmasını bu ağaç üzerinde çalıştıracak olursak, algoritma öncelikle derinlik değerini (bundan sonra d değişkeni (variable) ile ifade edilecektir) 0'dan başlatarak her adımda bir ilerletecektir.

d=0 için arama:
A
d=1 için arama:
A (tekrar A değerine bakar) B C (sanki ağacın en üstteki üç düğümü, şekilde gösterildiği gibi bağlanmış kabul edilebilir, daha alttaki derinliklerde bulunan DEFG düğümlerine hiç bakamaz)
d=2 için arama:
A B D E C F G (Bu aramada sanki sondaki daireler (cycle) bulunmuyormuş gibi kabul edilebilir, çünkü belirtilen derinlik bu dairelere kadar inemez)
d=3 için arama:
A B D B E C F G A
d=4 için arama:
A B D B D E B C F G A B C

Yukarıdaki arma işlemi, derinlik arttıkça devam etmektedir.

Arama algoritmalarının değerlendirildiği bazı kriterler bulunmaktadır. Buna göre IDDFS (iterative deepening depth first search) algoriması "tam" algoritma olarak kabul edilebilir. **Tamlık teriminin (completeness)** anlamı aşağıdaki şekilde tanımlanabilir:

Bir arama algoritması, bütün düğümleri dolaşarak aranan düğümü bulmayı garanti ediyorsa bu algoritmaya tam arama algoritması ismi verilir.

Örneğin yine yukarıdaki şekil için klasik derin öncelikli arama (depth first search) algoritmasını ele alsaydık, bu algoritmanın tam olmadığını söyleyebilirdik. Bunun sebebi DFS algoritmasının dolaşması sırasında, aşağıdaki döngüye girmesidir:

A B D B D B D (B D ikilisi sosuza kadar tekrar eder)

Kısacası A B D düğümleri dışındaki düğümlere asla bakmaz. Bu anlamda tam olmadığını söyleyebiliriz.

BFS (breadth first search , yayılma öncelikli veya sığ öncelikli arama) algoritması ise her zaman için tam kabul edilebilir, sebebi bir sonraki derinlik seviyesine inmeden önce, bulunduğu seviyedeki düğümleri bitirmesi ve bu sayede bütün ağaca bakmayı garanti etmesidir.

Algoritmanın karmaşıklığına değinecek olursak. Aşağıdaki şekilde bir formül ile karşılaşırız.

$$(d+1)1 + (d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d$$

Bu formülde görülen terimler şekil kuramında (graph theory) sıkça kullanılan bazı değerlere atıfta bulunur.
d değerinin derinlik olduğunu daha önce belirtmiştik. b ile gösterilen değer ise dallanma katasyısıdır (branching factor).

Kısaca her düğümün kaç çocuğu olduğu (out order, kaç düğüme gidilebildiği) olarak tutulur. Örneğin tam dolu ikili arama ağacının (binary search tree) b değeri 2'dir. Genelde en kötü durum analizinde bütün çocukların dolu olması ihtimali üzerinde durulur. Ayrıca istatistiksel olarak ortalamam çocuk sayısının da alındığı olur. Örneğin şehirlerin tutulduğu bir şekilde, her şehirden gidilebilecek şehir sayısı değişmektedir. Bu durumda b değeri ortalama değer olarak hesaplanabilir.

Bu açıklama ardından yukarıdaki denkleme tekrar bakarak anlamaya çalışalım.

IDDFS algoritmasında, en altta bulunan düğümlerin 1 kere (d sabitlendiğinde en altta kalan düğümler hangileri ise) dolaşıldığını, d-1 derinliğindeki düğümlerin ise 2 kere dolaşıldığını ve böylece kök düğüme (root) kadar her düğümün, bulunduğu seviyeye göre kaç kere dolaşıldığını hesaplayabileceğimize göre yukarıdaki denklem çıkarılabilir.

Diğer bir deyişle d=0 için kök 1 kere dolaşılırken d=1 olduğunda kök 2 ve kökün çocukları 1 kere dolaşılmış olur. d=3 olduğunda kök 3 kere ve çocukları 2 ve en alttaki çocuklar ise 1 kere dolaşılmış olur. Görüldüğü üzere derinliğin her artışında kök derinlik kadar altındaki her düğüm ise birer azalarak en nihayetinde yapraklar (leaf) tek bir kere dolaşılmış olmaktadır.

Sonuç olarak Yukarıdaki formülün ilk terimi kök (d+1)1 (tek düğüm d+1 kere dolaşılmıştır) , ikinci terimi kökün çocukları (db (kökün çocukları (ki sayısı b'dir) derinlik kadar (ki d ile gösterilir) dolaşılmıştır).

Yukarıdaki bu formülü daha toplu şekilde aşağıda gösterildiği gibi yazabiliriz:

$$\sum_{i=0}^{d}(d+1-i)b^i$$

Algoritmanın dolaşma (arama) zamanı karmaşıklığı yukarıda gösterildiği gibidir. Terimler birleştirildiğinde en kötü durum analizi (worst case analysis) için $O(b^d)$ gösterimi kullanılabilir.

Hafıza karmaşıklığı ise O(bd) olarak ifade edilebilir. Bunun sebebi d derinliğindeki bir ağacın her seviyesinde b adet çocuğu bulunması durumunda bd kadar düğüm olacağıdır. Algoritma, çalışması sırasında ilave düğümlere ihtiyaç duymaz ve şeklin kendisi üzerinde dolaşarak sonuca ulaşır.

SORU 7: Bin Packing (Kutulama Problemi)

İyileştirme problemleri açısından klasik bir örnektir (optimisation problems). Problem basitçe bir kutunun içerisine en az boş alan bırakarak, eşyaların en iyi şekilde nasıl yerleştireceği olarak düşünülebilir.

Aslında problemi boyutlara göre incelersek aşağıdaki şekilde bir liste yapılabilir:

Tek boyutlu kutulama (1D bin packing) :Bu problemde amaç bir çizgi veya hat gibi görülebilecek yapının içerisine farklı boyutlardaki çizgileri yerleştirmek olarak düşünülebilir.

Örneğin zaman çizelgelemesinde, bir kişinin yapacağı işleri, zaman çizgisinin üzerine yerleştirmesi (ve her işin farklı miktarda zaman gerektirdiğini ve kişinin çalışma saatlerinin sınırlı olduğunu düşünürsek en fazla işi en az zamanda (örneğin 8 saatlik mesailer içinde) yapması) bir problemdir. Buradaki hem kişinin yapacağı işler hem de bu işlerin yerleştirileceği zaman çizgisi tek boyutludur.

Daha basit olması açısından örneğin 100m uzunluğundaki bir ipi 7 ve 9m uzunluğundaki parçalara en az fire ile bölmek istiyoruz, en verimli bölme işleminde kaç adet 7 ve kaç adet 9 uzunluğunda ipimiz olur gibi soruları düşünebiliriz. Bu tip sorular tek boyutlu kutulama problemleridir.

İki boyutlu kutulama problemleri (2D bin packing optimization): Bu grupta bir tablodan ve iki boyuttan bahsedilebilir. Örneğin kot pantolon üreten bir tekstil firmasında farklı boyutlardaki Pantolon kalıplarının en az fire ile 5x5m büyüklüğündeki bir kare kumaştan kesilmesi isteniyor olsun. Bu problem iki boyutlu (x ve y boyutları) bir kutulama problemidir. Benzer bir problem, bir gazetedeki seri ilanların, en az fire ile sayfaya yerleştirilmesi olarak da düşünülebilir.

3 boyutlu kutulama (3D bin packing) problemin en zor şekli olarak tanımlanır ve 3 boyutlu bir kutunun içerisine konulan her şeklin farklı x,y ve z boyutlarında şekiller olması olarak düşünülebilir. Problemin genel tanımını yaptığımız için belirteyim, örneğin ev taşıma sırasında çıkan eşyaların kutulanması olarak düşünebilirsiniz. Bu durumu daha da karmaşık yapmaktadır çünkü kutular farklı boyut ve şekillerdedir (örneğin silindir bir varil veya küp veya dikdörtgenler prizması gibi kutuların içerisine yerleştirme yapılmakta) ve kutulanan şekillerde farklıdır ve hatta girintilidir (concave , non-convex) (örneğin avize, koltuk, sandalye gibi birbirinin içine girebilen nesneleri düşününüz). 2 ve 3 boyutlu paketleme, paketlenen nesnelerin girintili olup olmamasına göre ikiye ayrılmaktadır. Dış bükey nesnelerin paketlenmesi nispeten daha basit bir problemdir. Ancak nesnelerin iç bükey olması halinde problem biraz daha karmaşıklaşır.

Hatta literatürdeki kısıtlı aramalarım sonucunda ulaşabildiğim kadarıyla tam olarak iç bükey nesnelerin paketlenebildiği bir sonuç ne yazık ki bulamadım. Örneğin iki vidanın en verimlim paketlenmesi sırasında vidaların girinti çıkıntılarının üstüste gelmesi gerektiğini tecrübi olarak biliyoruz. N adet vida için bu durum birbirini tekrar eden bir hal alır. Gerçekten farklı boylarda ve adım sıklığında ve çaplarda vidalar verilse bu durumda en verimli paketlemeyi yapabilen bir algoritma henüz görmedim.

Paketlenen nesnelere göre problemin sınıflandırılması:

Paketlenen nesne çeşitlerinin sabit olması ve ön tanımlı olması halinde problem homojen olarak tanımlanır. Örneğin tek boyutlu kutulama probleminin tanımı sırasında verilen ve "100m uzunluğundaki bir ipi 7 ve 9m uzunluğundaki parçalara en az fire ile bölmek" şeklinde geçen örnek bu tip homojen (homogenous) bir yapıdadır. Buna karşılık heterojen bir problemde, paketlenecek nesnelerin tipleri ya tamamen birbirlerinden farklıdır ya da aynı tipte çok az tekrar vardır. Yine tek boyutlu problem örneğinde verilen zaman çizgisi üzerinde farklı uzunluklardaki randevuların yerleştirilmesi bu tiptendir.

Bu anlamda aşağıdaki problemler, kutulama probleminin birer özel hali olarak düşünülebilir:

- kamyon yükleme problemi (truck loading),
- konteyner yükleme problemi (Container loading problem, CLP)
- şerit paketleme problemi (Strip Packing problem, SPP)

Yukarıda, problemin tanımını yaptıktan sonra çözümlere bir göz atalım:

Homojen tek boyutlu problem çözümü

Şayet problem tek boyutlu ise ve homojen nesnelerin paketlenmesi olarak problemin çözülmesi isteniyorsa problem oldukça basit demektir ve basit matematiksel hesaplamalar ile problemi çözebiliriz.

Örneğin tek nesne ve tek paket varsa işlem basitçe paketin nesneye bölümü olarak bulunur (zaten burada zor Bir şey de yok):

Örneğin 100m uzunluğundaki bir ipten kaç tane 5m uzunluğunda ip kesilebilir:

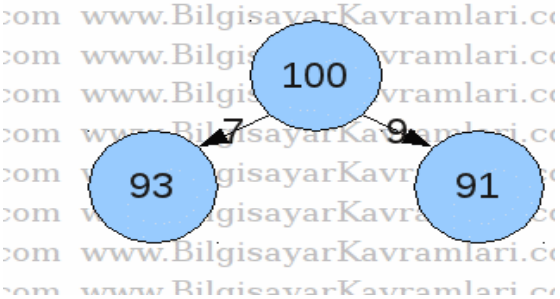
100 / 5 = 20

biraz daha zorlaştırıp ip sayısını 2 çeşide veya 3 çeşide çıkarırsak problem np-tam (np-complete) bir hal alır. Örneğin aşağıdaki kodu inceleyelim:

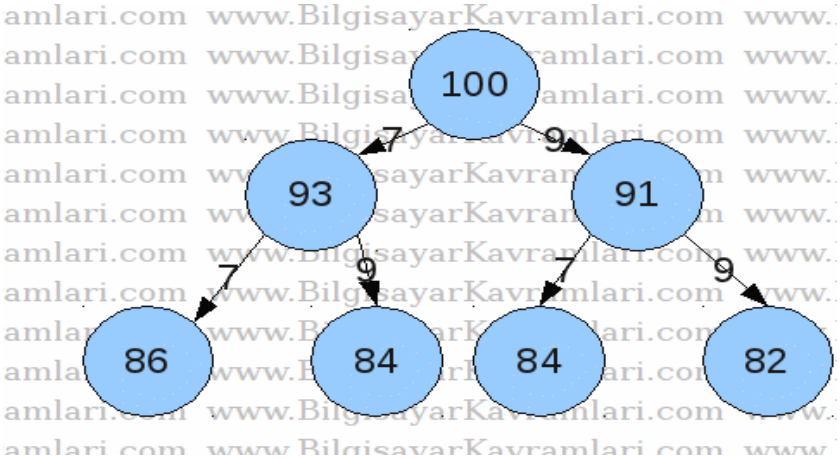
```c
#include <stdio.h>

int bul(int a,int b,int n,int m,int k){
  if(k==0){
    printf("\ncozum : %d*%d + %d*%d = %d",a,n,b,m,k);
    return 1;
  }
  if(k<0)
    return -1;
  int p = bul(a,b,n+1,m,k-a);
  int q = bul(a,b,n,m+1,k-b);
  if(p>q)
      return p;
  return q;
}
//www.bilgisayarkavramlari.com
int main(){
  int k = 100;
  int a = 7;
  int b = 9;
  while(bul(a,b,0,0,k)<0){
      printf("optimum cozum yok %d icin cozum deneniyor",--k);
  }
  return 0;
}
```

Kodda görüldüğü üzere bütün ihtimaller denenmektedir. Basitçe herhangi bir k değeri için, k-a ve k-b değerlerini denemekte ve denenen duruma göre a veya b değerini bir arttırmaktadır. Aslında kodumuz basit bir ikili ağaç (binary tree) oluşturmaktadır:

Önce 9 veya 7 ile başlanması ihtimalleri:



Sonra bu ihtimallerin de 7 veya 9 azalma ihtimalleri:



Yukarıda gösterildiği gibi her düğümden yine ikişer ihtimal indirerek bir alt seviyeye geçilebilir. Neticede 0 olana kadar yapılan bir aramadır ve 0 sonucuna birden farklı yoldan ulaşılabilir.

Kodumuz çalıştırıldığında çözüm olarak aşağıdaki sonuçları üretmektedir:

cozum : 7*4 + 9*8

cozum : 7*13 + 9*1

Gerçekten de problemin iki farklı çözümü bulunmaktadır.

Yukarıdaki kod basit bir hesaplama ile, 2′nin üstlerinin toplamı kadar adım hesaplamaktadır.

Bu değer yukarıdaki ağaçtan çıkarılabilir:

ilk düğüm için tek ihtimal 2^0

ikinci seviye için iki ihtimal: 2^1

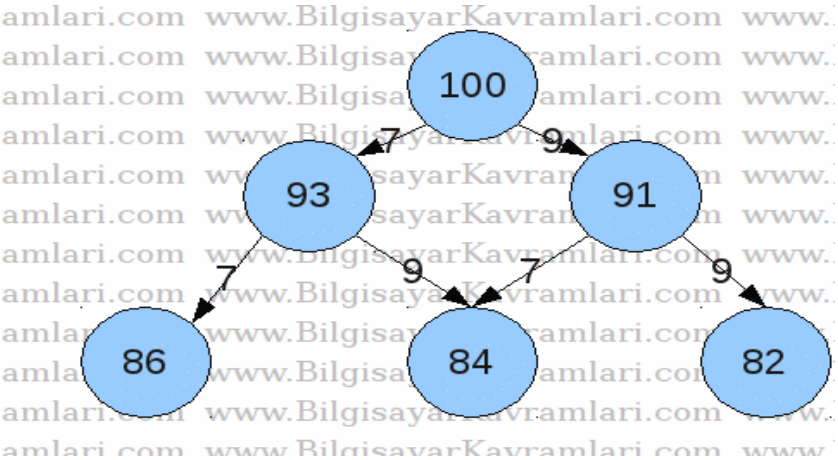
üçüncü seviye için dört ihtimal 2^2

şeklinde gitmektedir ve örneğin problemimiz üçüncü seviyede çözülseydi (sonuç 0 olsaydı) o zaman karmaşıklığımız bu değerlerin toplamı olacaktı ve $2^0 + 2^1 + 2^2$ şeklinde hesaplanacaktı.

Bu değer, ikili ağaçlardan bilindiği üzere 2^n-1 şeklinde hesaplanabilir.

Görüldüğü üzere yukarıdaki algoritma $O(2^n)$ değerinde bir karmaşıklığa sahiptir ve bu değer bir çok terimli (polynom) değildir yani algoritmanın karmaşıklık sınıfı np-tam (NP-Complete) olarak belirtilebilir. Ayrıca yukarıdaki k değeri için bir çözüm bulunmuştur ancak çözüm bulunamasaydı bu değer k terim için denenecekti. Yani 100 için çözüm yoksa bir yaklaşığı olan 99 için ardından iki yaklaşığı 98 için … Bu işlem hiç çözüm bulunamaması halinde k terim için denenecekti.

Yukarıdaki problem, dinamik programlama (dynamic programming) kullanılarak iyileştirilebilir. Bunun sebebi arama işlemi sırasında bazı sonuçların tekrar etmesidir. Örneğin yukarıdaki ikili ağacı aşağıdaki şekilde çizebiliriz:



Farka dikkat ederseniz, 93 – 9 = 84 ve aynı zamanda 91-7 = 84 olduğu görülür. Bu durumda aslında 84 değeri bir önceki kodda iki farklı durum için aranmaktayken şimdi tek bir durum için aransın istiyoruz. Elbette 84 sadece bir örnektir ve buna bağlı olarak çok sayıda tekrar eden değer bulunmaktadır.

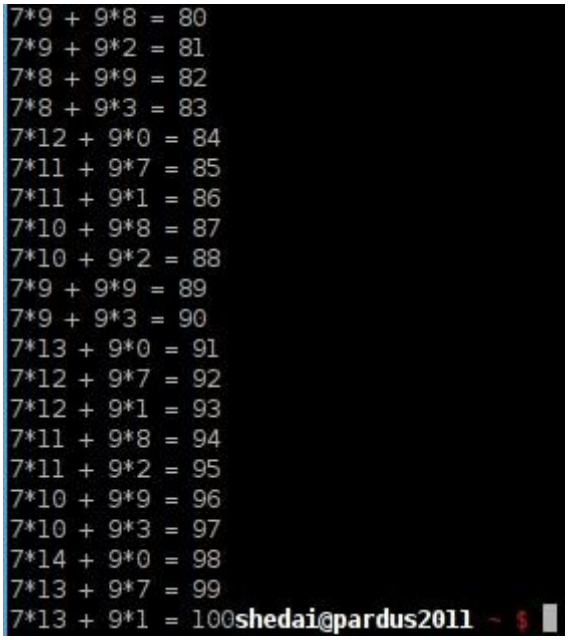
Hesaplanan bu değerleri bir dizi (array) içerisinde tutup tekrar hesaplanmasını engellemek istiyoruz:

```c
#include <stdio.h>
//www.bilgisayarkavramlari.com
int main(){
  int k = 100;
  int a = 7;
  int b = 9;
  int cozum[101][2]; // k = 100 icin 101
  for(int i = 0;i<=k;i++){
    cozum[i][0] = -1;
    cozum[i][1] = -1;
  }
  cozum[0][0] = 0; // 0 icin 0 tane 7
  cozum[0][1] = 0; // 0 icin 0 tane 9 kullanilir
  for(int i = a;i<=k;i++){
      if(cozum[i-a][0]>=0){
        cozum[i][0] = cozum[i-a][0]+1;
        cozum[i][1] = cozum[i-a][1];
      }
      else if(cozum[i-b][0]>=0){
        cozum[i][0] = cozum[i-b][0];
        cozum[i][1] = cozum[i-b][1]+1;
      }
  }
  for(int i = 0;i<=k;i++){
      printf("\n%d*%d + %d*%d = %d",a,cozum[i][0],b,cozum[i][1],i);
  }
  return 0;
}
```

Yukarıdaki yeni kodda, bir dizi içerisinde tek döngü ile sonucu hesaplattık. Buna göre algoritmamız iki elemanlı bir diziyi kullanmakta, dizinin 0. elemanları 7lerin sayısını ve 1. elemanları ile 9ların sayısını saymaktadır.

Kodumuz ilk başta 0 için 0 tane 7 ve 0 tane 9 gerektiği gerçeği ile çalışmaya başlıyor. 14-23. satırlar arasındaki döngü basitçe i. terim için i-a ve i-b değerlerine bakıyor. Şayet i. terim için i-a veya i-b değerinde bir çözüm varsa (nereden geldiğini önemsemeksizin) bu çözüme bulduğu koşulu ilave ederek mevcut i değeri için çözümü kaydediyor. Şayet bu iki terim de bulunmuyorsa o zaman bir sonraki i değerine geçiyor.

Ekran çıktısı aşağıdaki şekildedir:



Son 20 satır görülmekle birlikte daha önceki satılar alıntılanmamıştır. Ayrıca son 20 satırda görüldüğü üzere, tamamına ait bir çözüm bulunmaktadır. Örneğin 88 sayısı için 7*10 + 9*2 = 88 sonucuna ulaşılmıştır. En son satırda ise 100 için 7*13 + 9*1 sonucu görülmektedir.

Görüldüğü üzere birden fazla sonuç olsa bile tek bir sonucu görmekteyiz bunun sebebi veri yapısının bir sonuç üretmek üzere tasarlanmış olmasıdır. Elbette daha farklı veri yapıları kullanılarak diğer çözümleri de gösteren sonuçlar elde edilebilir.

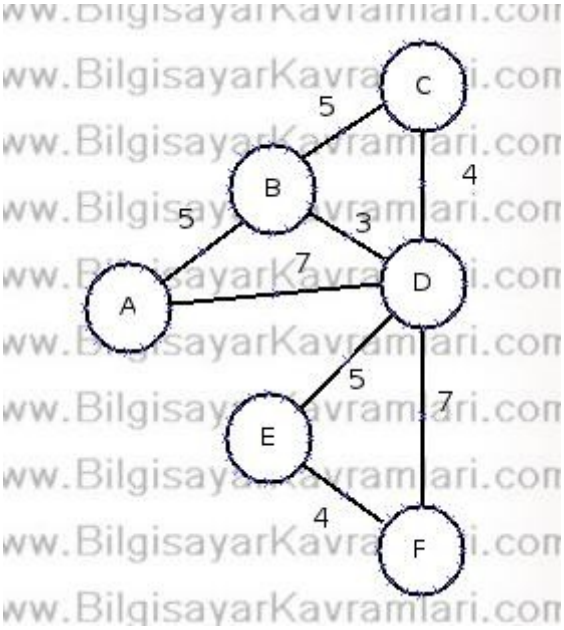
Yukarıdaki algoritmanın karmaşıklığı ise bir öncekine göre oldukça iyi sayılabilecek O(n) olarak bulunur. Bunun sebebi dizideki her elemanın üzerinden tek bir kere geçiyor olması ve dolayısıyla tek bir döngünün (koddaki 14-21 satırlar arası) çalışıyor olmasıdır.

SORU 8: Gomory-Hu Ağacı

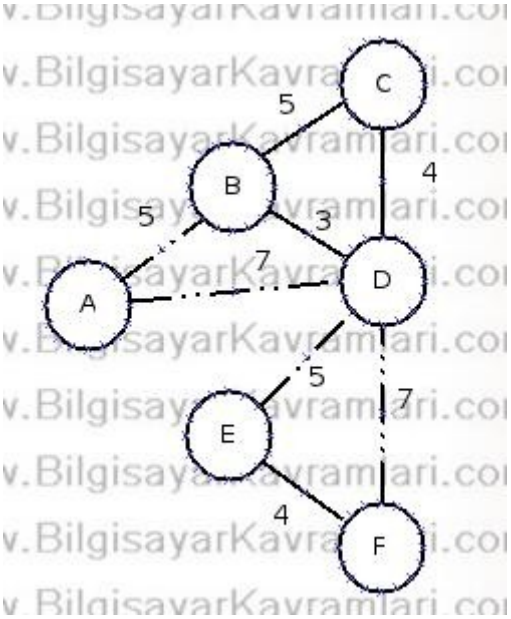
Bilgisayar bilimlerinde, şekil kuramında (graph theory) kullanılan en kısa kesim (minimum cut) problemine yönelik bir iyileştirme (optimization) ağacıdır.

Algoritmanın amacı, bir ağaç (tree) oluşturmak ve oluşturulan ağaçta, bir şekildeki (graph) kesme ihtimallerini hesaplamaktır.

Algoritmanın çalışmasını bir örnek üzerinden anlamaya çalışalım. Öncelikle en kısa kesimi bulacağımız şeklimiz (graph) aşağıdaki şekilde verilmiş olsun:



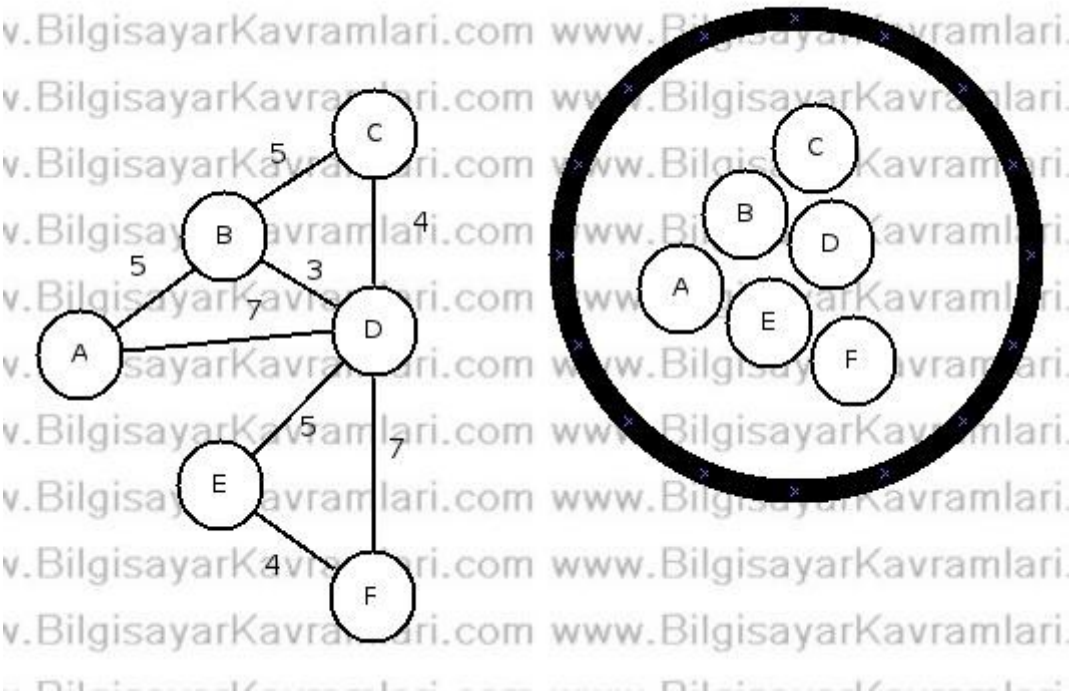
Yukarıda, yönsüz ve ağırlıklı bir şekil görülmektedir (undirected, weighted graph). Amacımız bu şekli düz bir şekilde kestiğimizde en az maliyetli kesiği bulmaktır. Konuyu anlamak için bir kesim örneği ele alalım:



Yukarıdaki kesim sırasında, A-B, A-D, E-D, F-D bağları kesilmiştir. Bu bağların toplam maliyeti 5+7+5+7 = 24 değerindedir.

İşte bizim amacımız farklı şekillerde yapılabilecek kesme eylemlerinden en az maliyete sahip olanını bulmaktır.

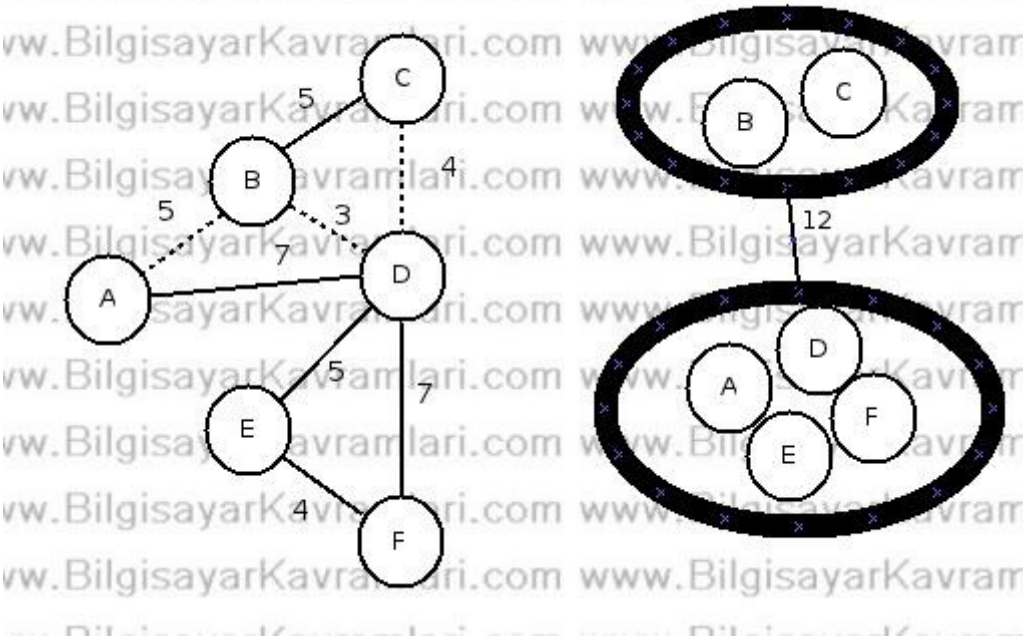
Algoritmamız ilk olarak bütün düğümlerin toparlandığı bir büyük düğüm ile başlıyor.



Şekilde görüldüğü üzere, bütün düğümleri içerisine alan tek bir düğümümüz bulunuyor. Şimdi rast gele seçilen iki düğüm arasındaki en küçük kesimi buluyoruz.

Bu aşamada rast gele olarak B ve E düğümlerini aldığımızı düşünelim.

Bu iki düğüm arasındaki en düğük maliyetli kesme aşağıda verilmiştir.



Yukarıdaki şekilde, görüldüğü üzere, A-B, B-D ve D-C kenarları kesilmiş ve bunun karşılığında 12 maliyetinde bir kesim elde edilmiştir. Algoritmamız, bu kesim işlemini düğümlerin tutulduğu grup üzerinde de uygular.

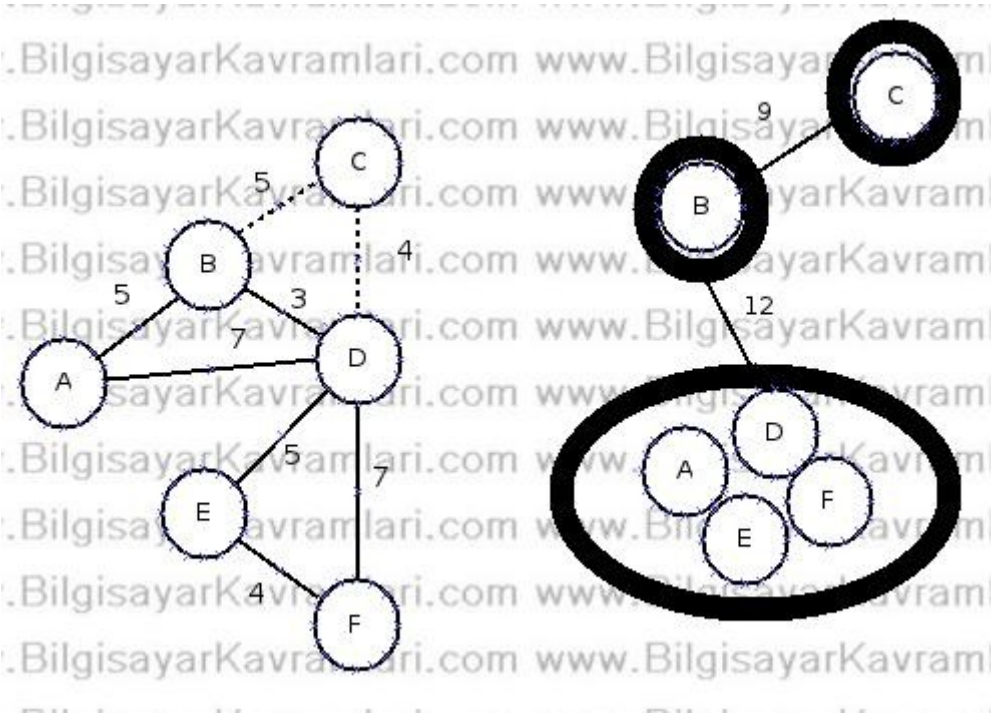
Aslında bu aşamada algoritma artık anlaşılmaktadır. Algoritmamız bu şekilde rast gele düğümler seçerek her grupta tek düğüm kalana kadar şekli parçalamaya devam edecek ve sonra en kısa maliyetli kenarı alacaktır.

Bu işlemi örnek üzerinde görmeye devam edelim.

Algoritmamızın şimdiki aşamasında, gruplardan birisini ve sonrada bu gruptaki düğümlerden iki tanesini seçeceğiz.

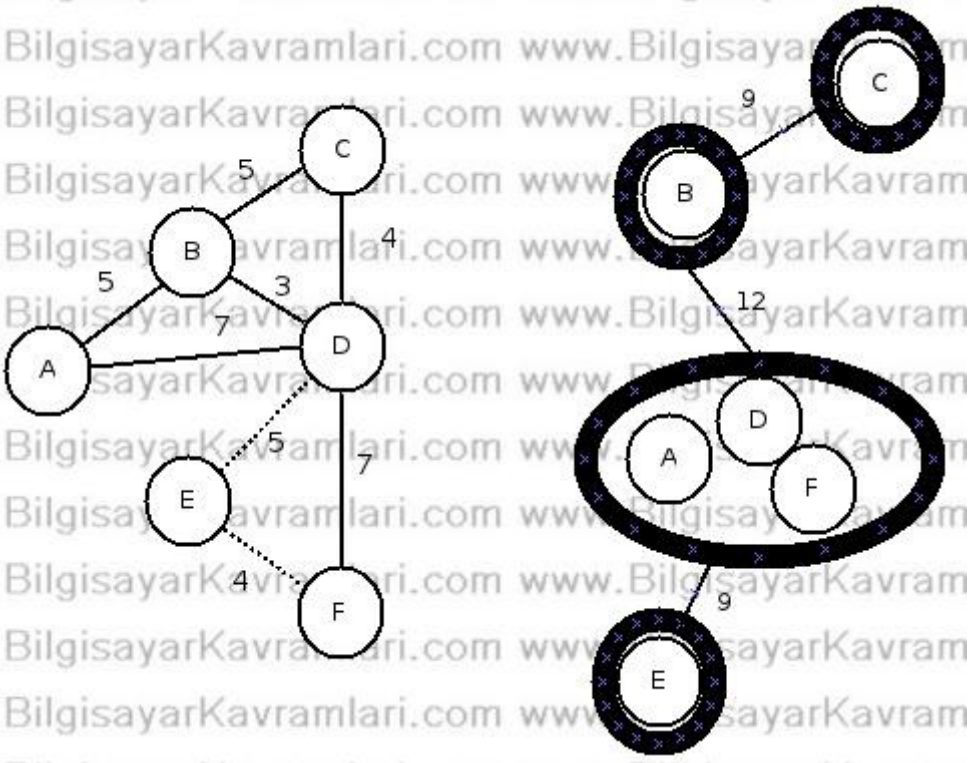
Örneğin B ve C düğümlerini seçtiğimizi düşünelim. (Bu aşamada farklı gruplardan iki düğüm seçemiyoruz. Örneğin B ve A düğümleri seçilemezdi. Ancak diğer gruptan iki düğüm seçebilirdik. Örneğin A ve D düğümleri olabilirdi).

B ve C düğümleri arasında tek bir kesme mümkündür ve bu da aşağıdaki şekilde işaretlenmiştir:



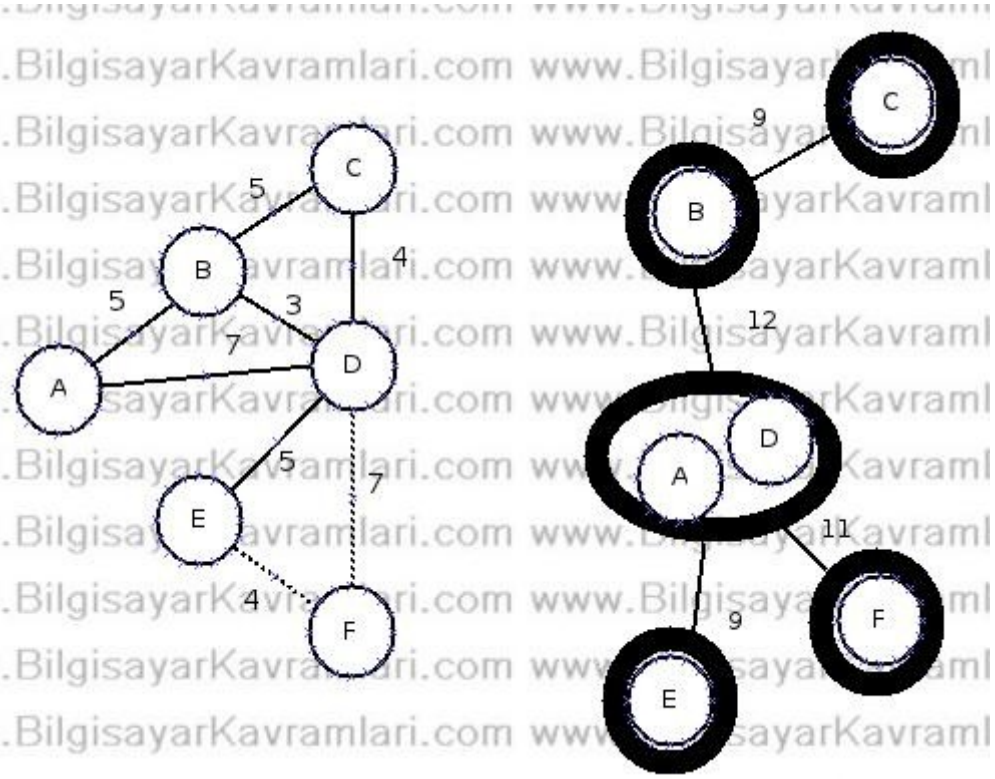
Yukarıdaki şekilde görüldüğü üzere, B-C ve C-D kenarları kesilmiş ve maliyet olarak 9 maliyetinde bir kesim elde edilmiştir. Bu durum algoritmamızın çalışmasını tutan resmin sağ tarafında gösterilmiştir. Yani bir önceki adımda beraber duran B ve C düğümleri ayrılmış ve iki ayrı düğüm olarak gomory-hu ağacında gösterilmiştir.

Bir sonraki adımda yine iki rast gele düğüm seçiyoruz. Bu sefer düğümlerimiz E ve F olsun.



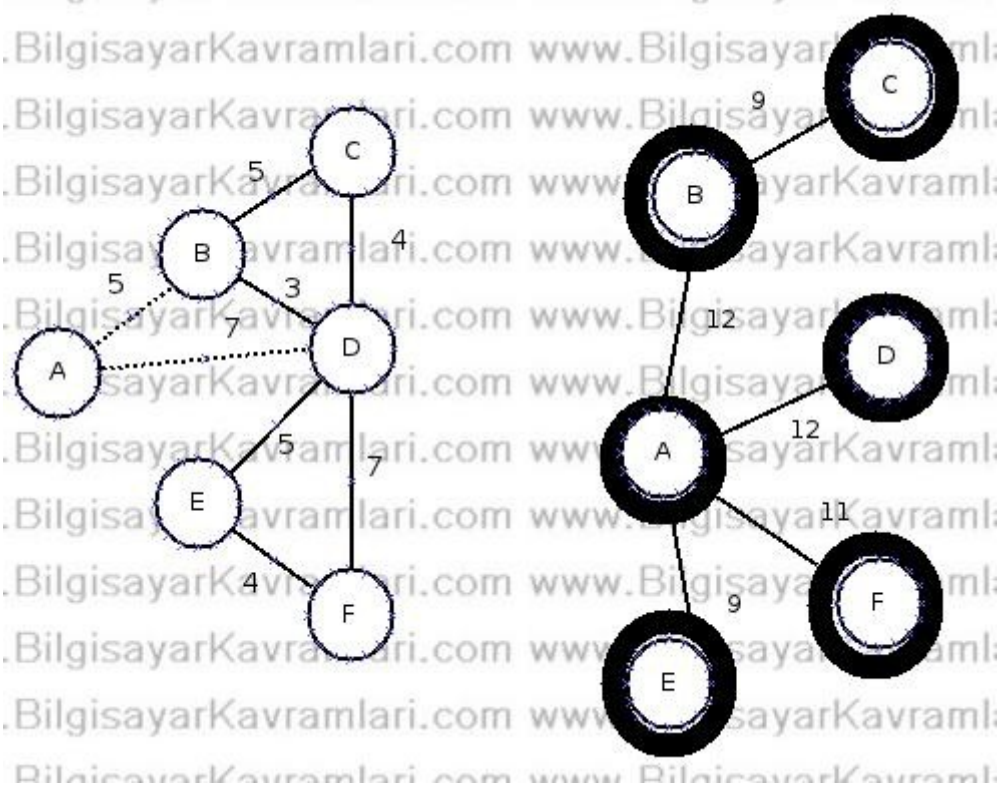
Yeni halimizde E düğümü de tek başına ağacımızda yerini almıştır (resmin sağ tarafı).

Rast gele düğüm seçimine devam ediyoruz ve bu sefer D ve F düğümlerini seçelim.



D ve F düğümleri arasında, iki farklı kesim alınabilirdi. En düğük maliyetli olan alternatif yukarıda zaten gösterilmiştir, ayrıca E-D ve F-D kesimi de alınabilirdi, ancak daha önce de belirttiğimiz üzere, en düşük maliyetli olanını alıyoruz.

Gelelim son ayrıma, A-D kesimi aşağıdaki şekilde yapılmıştır:



Bu son kesimle birlikte, ağacımızı tamamlamış oluyoruz ve algoritmamız çalışmasını bitiriyor. Buna göre yukarıdaki ağaçta iki tane minimum kesim (minimum cut) olabilir. Bunlardan birincisi B-C arasındaki 9 maliyetindeki kesim, ikincisi ise A-E arasındaki 9 maliyetindeki kesim.

SORU 9: Mealy ve Moore Makineleri (Mealy and Moore Machines)

Bilgisayar bilimlerinde sıkça kullanılan sonlu durum makinelerinin (finite state machine, FSM veya Finite State Automaton , FSA) gösteriminde kullanılan iki farklı yöntemdir. Genelde literatürde bir FSM'in gösteriminde en çok moore makinesi kullanılır. Bu iki yöntem (mealy ve moore makinaları) sonuçta bir gösterim farkı olduğu için bütün mealy gösterimlerinin moore ve bütün moore gösterimlerinin mealy gösterimine çevrilmesi mümkündür.

Klasik bir FSM'de bir giriş bir de çıkış bulunur (input / output). Bu değerlerin nereye yazılacağı aslında iki makine arasındaki farkı belirler.

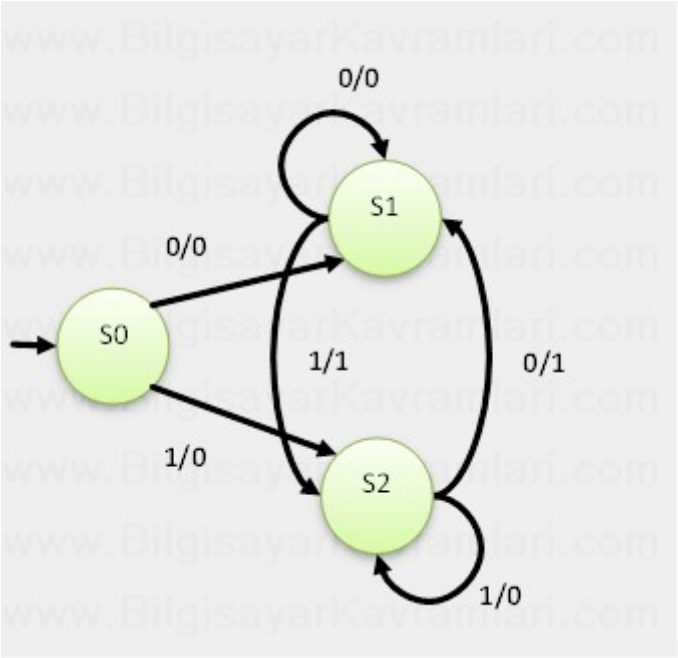
Moore makinelerinde çıkış değerleri düğümlere (node) yazılırken, giriş değerleri kenarlar (edges) üzerinde gösterilir.

Mealy makinelerinde ise giriş ve çıkış değerleri kenarlar (edges) üzerinde aralarına bir taksim işareti (slash) konularak gösterilir. Örneğin 1/0 gösterimi, girişin 1 ve çıktının 0 olduğunu ifade eder.

Basit bir örnek olarak özel veya (exclusive or, ⊻) işlemini ele alalım ve her iki makine gösterimi ile de çizmeye çalışalım.

Girdi 1	Girdi 2	Çıktı
0	0	0
0	1	1
1	0	1
1	1	0

Klasik bir XOR kapısı, yukarıdaki doğruluk çizelgesinde (truth table) gösterildiği üzere iki giriş ve bir çıkıştan oluşur. Bizim makinemiz de ilk girdiden sonra ikinci girdiyi aldığında beklenen çıktıyı vermeli. Ayrıca makine sürekli olarak çalışmaya devam edecek. Örneğin 101011 şeklinde bir veri akışı sağlanması durumunda, sonuç olarak 1⊻0⊻1⊻0⊻1⊻1 değerini hesaplamasını isteriz.



Yukarıdaki gösterim bir mealy makinesidir. Makinede görüldüğü üzere 3 farklı durum arasındaki geçişler üzerine iki adet değer yazılmıştır. Bu değerlerden ilki giriş ikincisi ise çıkış değeridir.

Makineyi beraber okumaya çalışalım.

Makinenin boşluktan bir ok ile başlayan durumu, yani örneğimizdeki S0 durumu, başlangıç durumudur. Bu durumdan başlanarak gelen değerlere göre ilgili duruma geçilir.

Örneğimizi hatırlayalım. Giriş olarak 101011 dizgisini (string) almayı planlamıştık. Bu durumda ilk bitimiz 1 olarak geliyor ve S0 durumunda 1 girişi ile S2 durumuna geçiyoruz. Burada geçiş sırasında kullanılan kenarın üzerindeki değeri okuyalım: 1/0 bunun anlamı 1 geldiğinde geçilecek kenar olması ve çıktının 0 olmasıdır. Yani şu anda çıktımız 0

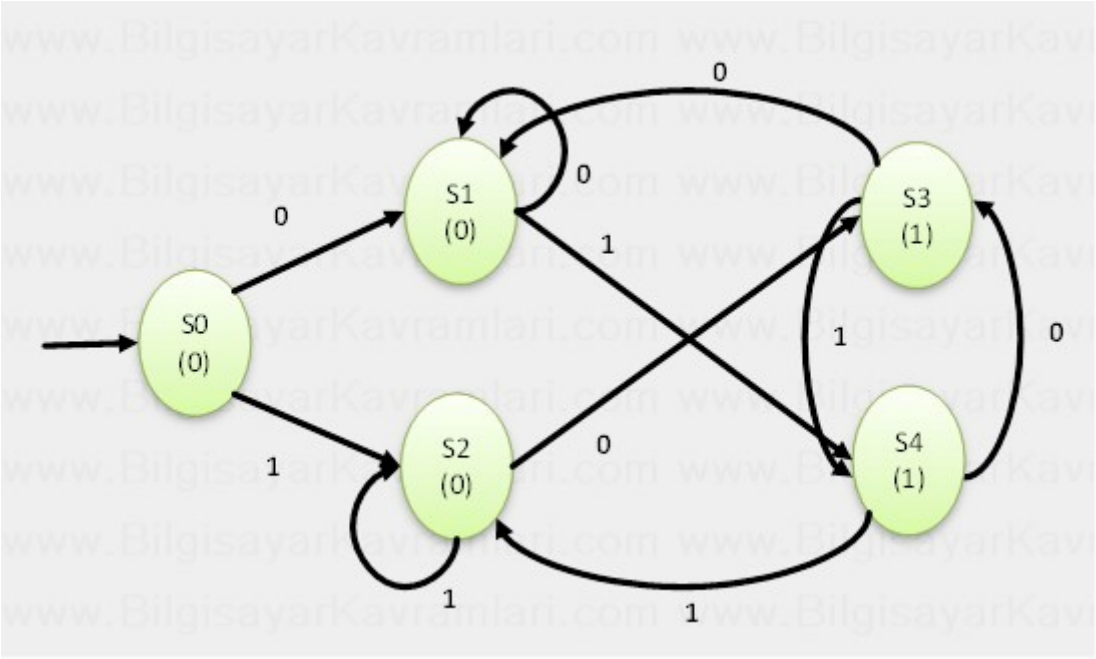
Ardından gelen değer 0 (yani şimdiye kadar 01 değerleri geldi). En son makinemizdeki durum, S2 durumuydu, şimdi yeni gelen değeri bu durumdaki kollardan takip ediyor ve S1′e giden 0/1 kenarını izliyoruz. Bu kolu izleme sebebimiz, S2 durumundan gidilen tek 0 girdisi kolu olmasıdır. Bu kol üzerindeki ikinci değer olan 1 ise, çıktının 1 olduğudur. Yani buraya kadar olan girdiyi alacak olsaydık 01 için 1 çıktısı alacaktık.

İşlemlere devam edelim ve durumları yazmaya çalışalım:

Gelen Değer	Durum	Çıkış	Yeni Durum
1	S0	0	S2
0	S2	1	S1
1	S1	1	S2
0	S2	1	S1
1	S1	1	S2
1	S2	0	S2

Yukarıdaki tablomuzun son halinde, çıkış değeri olarak 0 okunmuştur. Yani örneğimizin neticesi 0 olacaktır.

Aynı örneği moore makinesi olarak tasarlayacak olsaydık:



Moore makinesinde, mealy makinesine benzer şekilde boşluktan gelen bir ok, başlangıç durumunu belirtir.

Makinenin, durumlarında, mealy makinesinde olmayan değerler eklenmiştir. Bu değerler, ilgili durumdaki çıktıyı gösterir. Örneğin makinemiz S1 durumundayken çıktı 0 olarak okunabilir.

Şimdi moore makinesinde, aynı örneği çalıştırıp sonucu karşılaştıralım.

Giriş olarak 101011 dizgisini (string) almayı planlamıştık. Bu durumda ilk bitimiz 1 olarak geliyor ve S0 durumunda 1 girişi ile S2 durumuna geçiyoruz. Bu geçiş sonucunda geldiğimiz S2 durumunda okunan çıktı değeri 0 yani sonuç şimdilik 0.

Ardından gelen 0 değeri ile S3 durumuna geçiyoruz ve çıktımız 1 oluyor. Çünkü S3 durumu 1 çıktısı veren durumdur. Bu şekilde durumları ve durumlar arasındaki geçişleri izlersek, aşağıdaki tabloyu çıkarabiliriz:

Gelen Değer	Durum	Çıkış	Yeni Durum
1	S0	0	S2
0	S2	1	S3
1	S3	1	S4
0	S4	1	S3
1	S3	1	S4
1	S4	0	S2

Görüldüğü üzere, makinemiz, örnek girdi için S2 durumunda sonlanıyor ve bu durumda çıktımız 0 olarak okunuyor.

SORU 10: Order Theory (Sıra Teorisi, Nazariyatül Tertib)

Bu yazının amacı, eğitimimiz sırasında sürekli olarak okuduğumuz bir teori olan tertip teorisi (sıralama teorisi , order theory) konusunda bulunan kavramları (preorder, postorder gibi) açıklamaktır. Osmanlıda bu konuya nazariyatül tertip ismi verilmektedir.

Nazariyatül tertip, bilgisayar bilimleri de dahil olmak üzere pek çok matematiksel problemin çözümünde kullanılmaktadır. Örneğin sıralama algoritmaları, arama algoritmaları gibi basit algoritmalar, bu nazariye üzerinden çalışmaktadır.

Her şeye en basitten başlayalım ve ilk okulda öğrendiğimiz < veya > işaretlerinin aslında bu nazariyenin en temel işlemleri olduğunu belirtelim. Buna göre biz tam sayılar arasında bir sıralama (tertip) işlemi yaparken aslında bu teorinin bir uygulamasını yapıyoruz.

Örneğin aşağıdaki tanımı ele alalım:

Partially Ordered Sets (kısmi tertipli kümeler)

Bir kümenin aşağıdaki özellikleri taşıması durumunda verilen isimdir:

$a \leq a$ (yansıma (reflexive) özelliği)

$a \leq b$ ve $b \leq a$ ise $a = b$ (ters simetri özelliği (antisymmetry))

$a \leq b$ ve $b \leq c$ ise $a \leq c$ (geçişlilik (transitivity)).

Örneğin yukarıdaki özelliklerin hepsi sayma sayıları kümesi için geçerlidir.

Bu kümeye İngilizcedeki partially ordered sets isminin ilk iki kelimesini birleştirerek poset ismi de verilir.

Preorder (Öntertip)

Bir değerin diğerinden önce gelmesi prensibine dayanır. Öntertip şartının sağlanması için, bir serideki değerler arasında yansıma (reflexive) ve geçişlilik (transitivity) özellikleri bulunmalıdır. Ancak ters simetri (Anti symmetry) bulunmak zorunda değildir.

Buna göre sadece a < b < c şeklindeki bir seri, öntertip olarak kabul edilebilir ve bu bağlantı özelliklerinde anti simetriden gelen eşitlik bulunmak zorunda değildir.

İşlem Tertipleri

Matematiksel olarak bütün gösterimleri bir dil olarak kabul etmek mümkündür. Bu anlamda işlemleri ifade eden matematiksel gösterimler de birer dildir ve bazı kurallara tabidir. Matematiksel işlemlerin gösteriminde kullanılan yazım şekilleri farklılık gösterebilir.

Örneğin klasik olarak kullanılan matematiksel işlemler, ⊶ herhangi bir işlem olmak ve a ve b birer fiil (predicate) olmak üzere, işlemler aşağıdaki şekillerde gösterilebilir:

1. Inorder (iç tertip infix): a ⊶ b
2. Preorder (Ön tertip, prefix): ⊶ a b
3. Postorder (Son tertip, postfix): a b ⊶

Yukarıdaki yazılımlardan iç tertip gösterimi, ülkemizde ilk okullardan beri öğretilen gösterimdir. Örneğin ” 5 + 6 ” gösteriminde + işlemi iki değerin arasında yer almaktadır.

“+ 5 6″ ön tertbi veya “5 6 +” son tertibi de çeşitli matematiksel gösterimlerde kullanılmaktadır.

Örneğin programlama dillerinde de sıkça kullanılan fonksiyon çağırımı bir ön tertiptir.

F 4 5

Gösteriminde, F fonksiyonuna 4 ve 5 değerleri birer parametre olarak verilmiştir. Benzer bir kullanım komut satırında da programlara parametre verilirken geçerlidir.

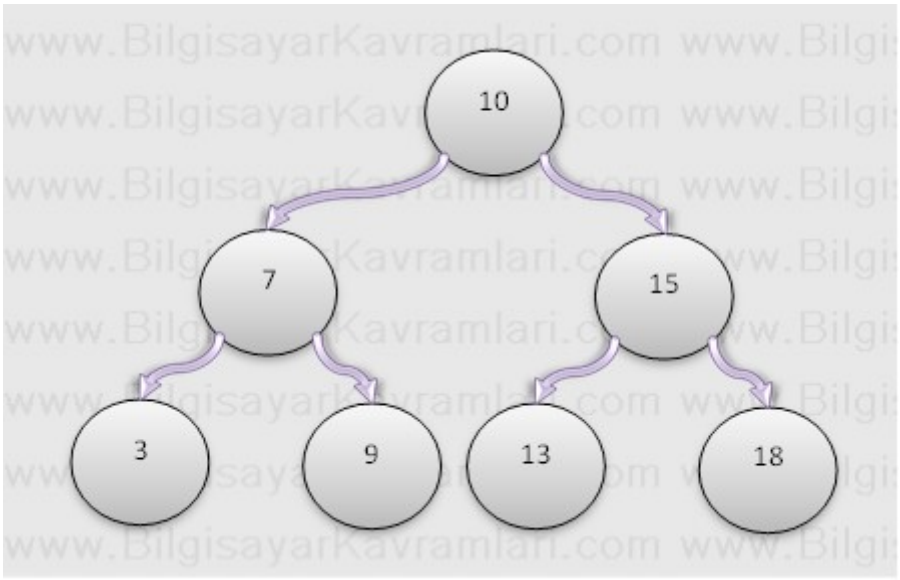
cp a.txt b.txt

Yukarıdaki gösterimde a.txt dosyası, b.txt dosyası olarak kopyalanmış bu sırada, işleme (cp) parametreler prefix (öntertip) sırasıyla verilmiştir.

Ağaç Dolaşma Algoritmaları (Tree Traverse Algorithms)

Sıralama teorisinin diğer bir kullanım alanı da ağaçlardır. Herhangi bir ağacın dolaşılması sırasında yukarıda açıklanan sıralamalar kullanılabilir.

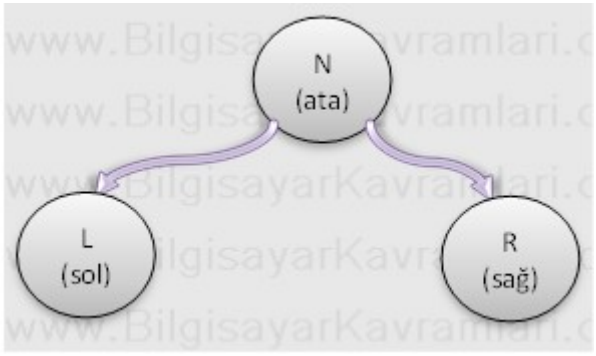
Konuyu en çok kullanılan ikili ağaçlar (binary tree) üzerinden açıklamaya çalışalım.



Yukarıdaki ağaç örneği, ikili ağaç olmanın yanında, ikili bir arama ağacı (binary search tree) olma özelliğini de taşır.

Buna göre her düğümün solundaki düğümler, kendisinden küçük ve sağındaki düğümler ise kendinden büyük değerler taşımak zorundadır.

Yukarıdaki herhangi bir düğümü aldığımızda aşağıdaki gibi bir durum ortaya çıkar.



Yukarıdaki gösterim, ikili arama ağacının bütün düğümleri için geçerlidir. Belki yaprak düğümler (leaf nodes, ağacın en altındaki düğümler) için bu durumun geçerli olmadığını düşünebilirsiniz ancak, en sondaki bu düğümlerin de sol ve sağında alt düğümler bulunmakta ancak bu düğümlerin değeri boş (null) olmaktadır. (daha detaylı bilgi için ikili arama ağacının kodlamasına bakabilirsiniz)

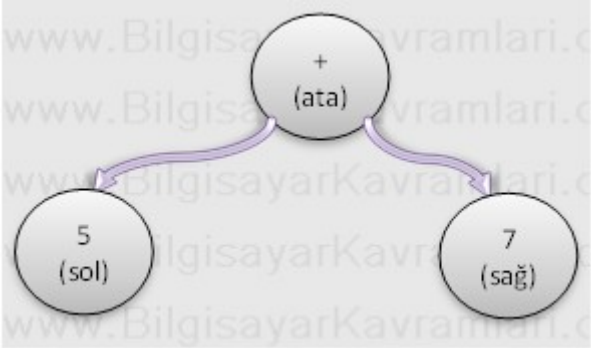
İkili arama ağaçlarında 6 farklı dolaşım mümkündür. Zaten bu değer 3!'dir. Bu dolaşımları yukarıda modellediğimi 3 düğümden oluşan ağaç için sıralayalım:

1. NLR
2. NRL
3. LNR
4. RNL
5. LRN
6. RLN

Yukarıdaki gösterimleri, N (ata) düğümün konumuna göre 3 sınıfta toplamak mümkündür.

1. Infix (iç tertip , inorder) : LNR ve RNL
2. Prefix (öntertip , preorder): NLR ve NRL
3. Postfix (sontertip, postorder): RLN ve LRN

Yukarıdaki bu gösterimleri birer işlem ağacı olarak da düşünmek mümkündür. Örneğimizi aşağıda çizerek anlatmaya çalışalım:



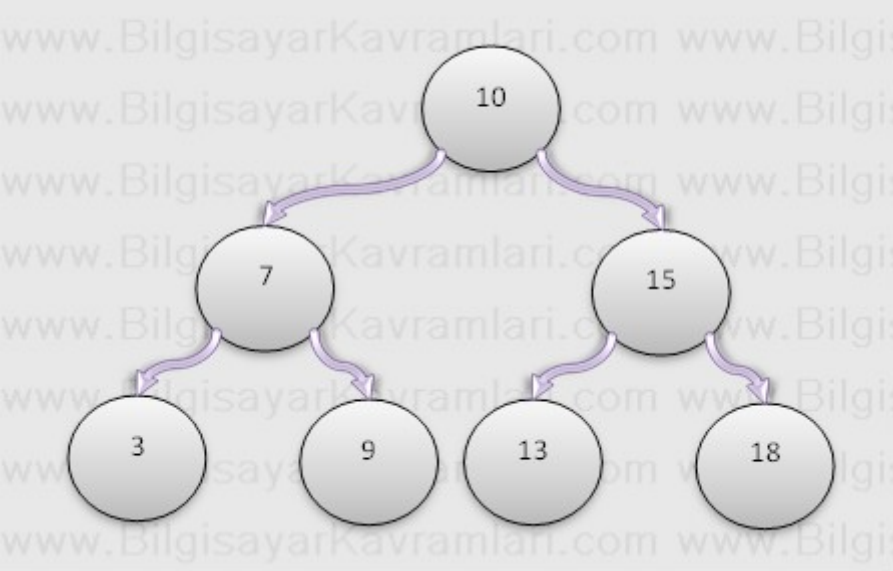
Yukarıdaki ağaçta bulunan değerleri 6 farklı gösterim için okuyalım:

1. NLR : + 5 7
2. NRL : + 7 5
3. LNR : 5 + 7
4. RNL : 7 + 5
5. LRN : 5 7 +
6. RLN : 7 5 +

Bu gösterimlerden örneğin LNR gösterimi bizim klasik olarak ilk okuldan beri gördüğümüz toplama işlemini ifade eder.

Buna karşılık RNL gösterimi ise işlemin tersini ifade etmektedir ve toplama örneği için doğru olmakla birlikte, çıkarma (-) işleminde LNR ve RNL sırası fark etmektedir.

Şimdi örnek olarak aldığımız ilk ağaca dönelim ve 6 farklı dolaşmaya göre sayıları listeleyelim:



1. NLR : 10 7 3 9 15 13 18
2. NRL : 10 15 18 13 7 9 3
3. LNR : 3 7 9 10 13 15 18
4. RNL : 18 15 13 10 9 7 3
5. LRN : 3 9 7 13 18 15 10
6. RLN : 18 13 15 9 3 7 10

Yukarıdaki ağaç dolaşma algoritmalarından, 1. Ve 2. Sırada olanları (ikisi de öntertiptir (preorder)) literatürde, önce düğüme sonra altındaki düğümlere baktığı için sığ öncelikli dolaşma / arama (breadth first search) olarak isimlendirilir.

Buna karşılık diğer 4 dolaşma sıralaması, önce derindeki bir düğüme sonra ataya baktığı için derin öncelikli (depth first search) olarak geçer.

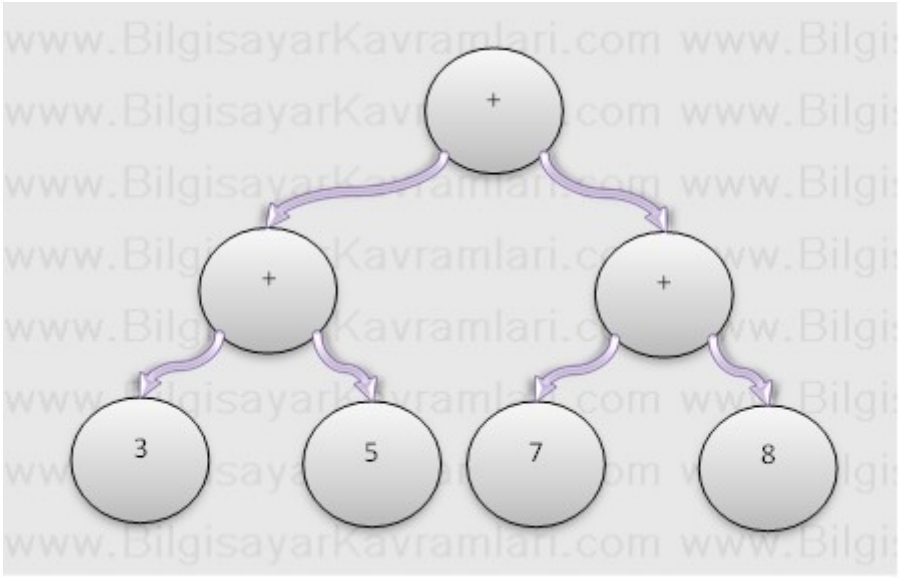
Yine yukarıdaki sıralamalarda dikkat edilecek bir huşu, LNR gösteriminin, ağacı küçükten büyüğe, RNL gösteriminin ise ağacı büyükten küçüğe sıraladığıdır. Bu özellik, ağaç sıralaması (tree sort) olarak geçen sıralama algoritmasının temelini oluşturur.

İşlem Tertiplerinin Dönüşümü

Örneğin elimizde içtertip (infix) bir işlem bulunsun ve biz bu tertibi örneğin son tertip (postfix) olarak çevirmek isteyelim:

(3 + 5) + (7 + 8)

Öncelikle yukarıdaki işlemin parçalama ağacını (parse tree) oluşturuyoruz.



Yukarıdaki ağaçta, işlem önceliğine göre parçalama yapılmıştır. Örneğin 3 ve 5 sayılarını birleştiren işlem + işlemidir. Benzer şekilde 7 ve 8 sayıları diğer bir toplama işlemi ile birleştirilmiş ve neticede (kökte) her iki işlem toplanmıştır.

Şimdi bu ağacı öğrendiğimiz ön tertip yöntemine göre dolaşalım:

NLR : + + 3 5 + 7 8

Yukarıdaki gösterim, verilen işlemin son tertibe çevrilmiş halidir. Benzer durum son tertip için de geçerlidir:

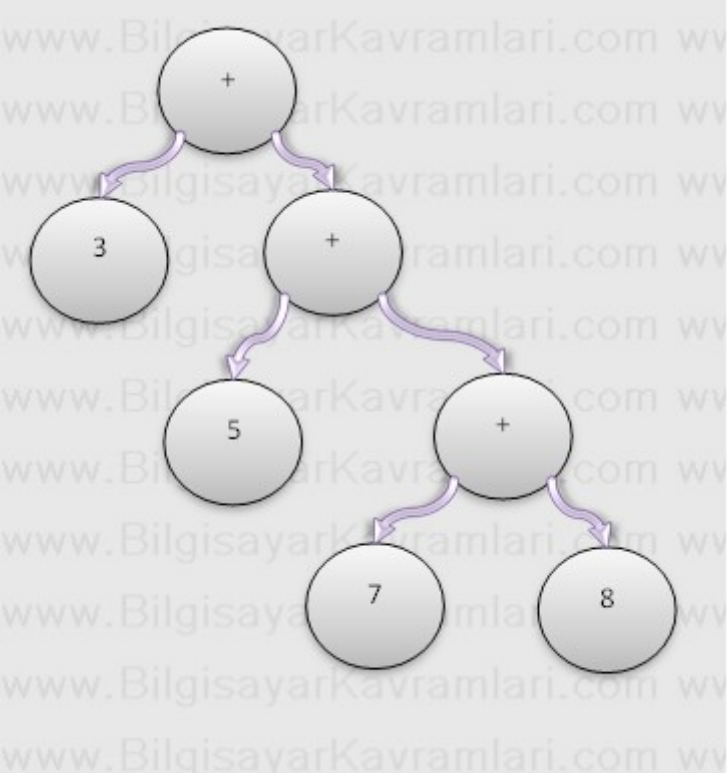
LRN : 3 5 + 7 8 + +

Şayet örneğimiz aşağıdaki şekilde olsaydı:

3 + 5 + 7 + 8

Bu durumda parçalama ağacında bir belirsizlik oluşacaktı (ambiguity).

Örneğin, yukarıdaki bu yeni işlem, yukarıdaki parçalama ağacı gibi parçalanabilirken aynı zamanda aşağıdaki ağaçlardan birisi olarak da parçalabilir:



Veya

Yukarıdaki her iki gösterim de aslında matematiksel olarak işlemden çıkarılabilecek sonuçlardır.

Örneğin yukarıdaki işlemi şu 3 farklı şekilde parantezlere almak (veya 3 farklı sırada çözmek mümkündür)

(3+5)+(7+8)

3+(5+(7+8))

((3+5)+7)+8

Bu üç yaklaşımda doğrudur. Bu durumda, işlemin ön tertip çevirimi 3 farklı şekilde olabilecektir ve 3ü de doğrudur:

(3+5)+(7+8) → + + 3 5 + 7 8

3+(5+(7+8)) → + + + 7 8 5 3

((3+5)+7)+8 → + + + 3 5 7 8

Benzer durum son tertip için de geçerlidir:

(3+5)+(7+8) → 3 5 + 7 8 + +

3+(5+(7+8)) → 3 5 7 8 + + +

((3+5)+7)+8 → 8 7 3 5 + + +

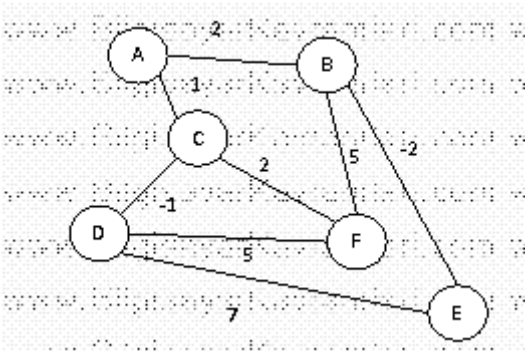
Yukarıdaki bu dönüşümlerden görüleceği üzere artık belirsizlik durumu kalkmıştır. Yani dönüşümlerden herhangi birisinin tercih edilmesi halinde iç tertip durumundaki belirsizlik kaybedilir. Bu anlamda tam bir dönüşüm olduğunu söylemek mümkün olmaz.

SORU 11: Bellman Ford Algoritması

Bu algoritmanın amacı, bir şekil (graph) üzerindeki, bir kaynaktan (source) bir hedefe(target veya sink) giden en kısa yolu bulmaktır. Algoritma ağırlıklı şekiller (weighted graph) üzerinde çalışır ve bir anlamda Dijkstra algoritmasının iyileştirilmişi olarak düşünülebilir.

Algoritma aslında Dijkstra algoritmasından daha kötü bir performansa sahiptir ancak graftaki ağırlıkların eksi olması durumunda Dijkstra'nın tersine başarılı çalışır.

Algoritma Dijkstra algoritmasında olduğu gibi en küçük değere sahip olan kenardan gitmek yerine bütün graf üzerindeki kenarları test eder. Bu sayede aç gözlü yaklaşımının (greedy approach) handikabına düşmez ve her düğüme sadece bir kere bakarak en kısa yolu bulmuş olur.

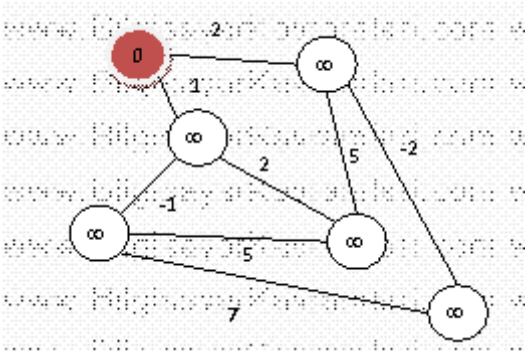


[flashvideo file=http://www.bilgisayarkavramlari.com/wp-content/uploads/bellmanford.flv /]

Algoritmanın çalışmasını yukarıdaki gibi eksi değerlere sahip bir şekil üzerinden anlamaya çalışalım.

Öncelikle düğümlere değer ataması yapılıyor. Başlangıç düğümüne 0, doğrudan erişilen düğümlere erişim değerleri ve erişilemeyen düğümlere 👈 sonsuz değeri atanıyor. Hedef düğüm olarak da E düğümünü tanımlayalım.

Yukarıdaki örnekte A düğümünden başlayacağımız düşünelim:



Şimdi şekilde kenarları sıralayalım:

AB 2

AC 1

BE -2

BF 5

CF 2

CD -1

DF 5

DE 7

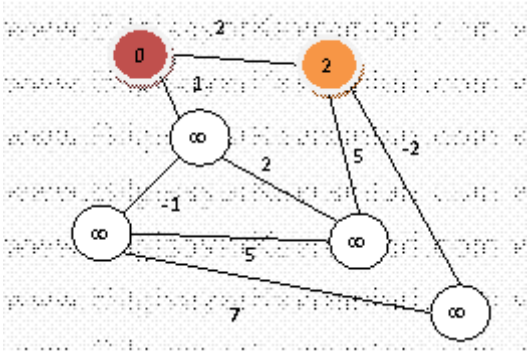
Bellman ford algoritması işte bu kenarları teker teker dolaşması itibariyle dijkstradan ayrılır. Sırayla yukarıdaki kenarları (Edges) dolaşır ve graftaki değerleri günceller.

Yukarıda, rastgele sıra ile dizilen bu kenarları sırasıyla dolaşalım ve bu dolaşma sırasında şu işlemi yapalım.

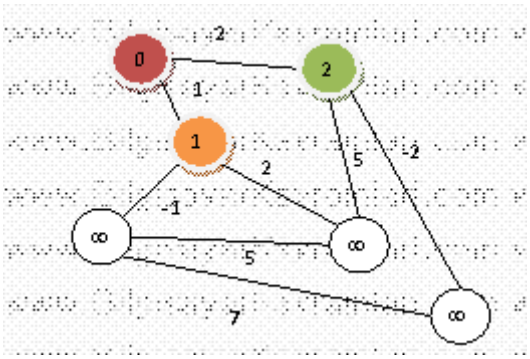
Örneğin AB kenarı , A ve B düğümleri arasında. Bu düğümlerden düşük değere sahip olan düğüm üzerine kenar değeri eklenip yüksek değere sahip olan düğüm üzerine yazılıyor.

Sırayla gidecek olursak:

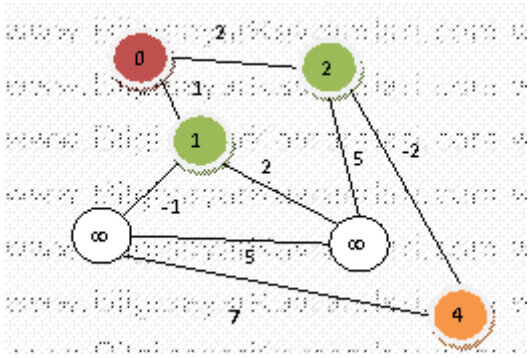
AB 2, min(A,B) = 0 ✉ 0+ 2 = 2



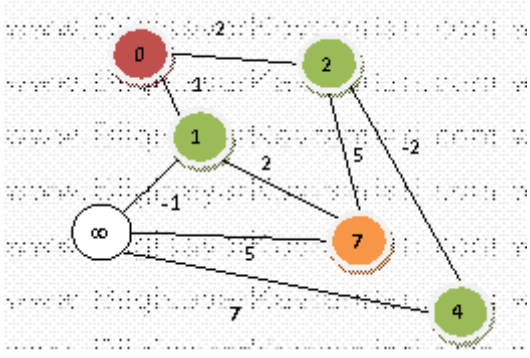
AC 1, min(A,C) = 0 => 0+1 = 1



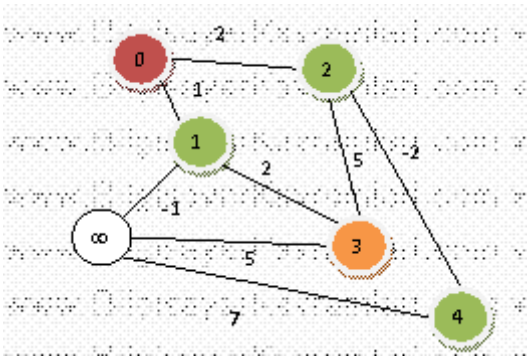
BE 2, min (B,E) 2, => 2 + 2 = 4



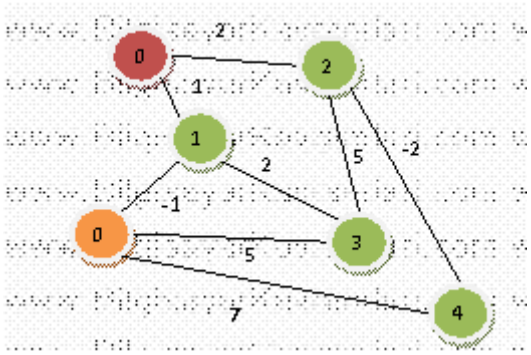
BF 5, min(B,F) = 2 => 2+5 = 7



CF 2, min (C,F) = 1 => 1+2 = 3, bu değer 7′den küçük olduğu için güncelliyoruz:



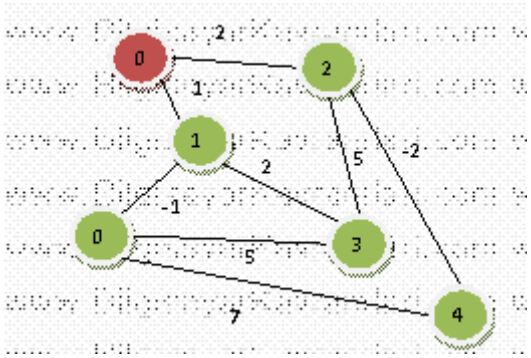
CD -1, min(C,D) = 1=> 1+(-1) = 0



DF 5, min (D,F) = 0 => 0+5 = 5 F'nin değerinden daha büyük o yüzden güncellemiyoruz.

DE 7, min(D,E) = 0 => 0+7 = 7, yine E'nin değerinden büyük olduğu için güncellenmiyor.

Sonuçta graf aşağıdaki şekilde bitmiştir:



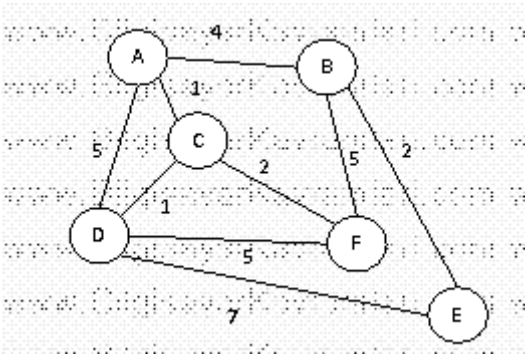
Bu graftaki bütün düğümlerde, A düğümünden ne kadar maliyetle gidildiği bulunmuştur. Örnek hedef E düğümü ise, ulaşım maliyeti 4′dür.

Görüldüğü üzere eksi değere sahip düğümler olmasına rağmen herhangi bir problem yaşanmamıştır.

SORU 12: Edmonds Karp Algoritması

Bu algoritmanın amacı, literatürde azami akış (maximum flow) olarak geçen ve düğümler (nodes) arasında akış kapasiteleri belirli bir şekildeki (graph) bir başlangıçtan bir hedefe en fazla akışın sağlandığı problemleri çözmektir.

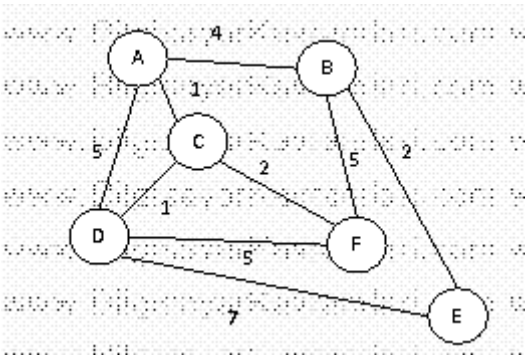
Azami akış (maximum flow) problemini örneğin şehirler arasında bağlı boru hattına veya tedarik zincirine benzetebiliriz. Örneğin aşağıdaki şekli ele alalım:



Buradaki düğümler, şehirleri ve düğümler arasındaki kenarlar (edges) ise şehirler arasındaki boru hatlarının kapasitesini belirtsin. Amacımız A düğümünden E düğüme azami miktarda akış sağlayabilmek olsun.

Ford-Fulkerson çözüm için yukarıdaki şekilde öncelikle hedef düğüme giden yolu bulur. Algoritma bu arama işlemi sırasında şayet derin öncelikli arama (depth first search ,DFS) kullanıyorsa ford fulkerson olarak isimlendirilir. Şayet aynı algoritma bu arama işlemi sırasında sığ öncelikli arama (breadth first search, BFS) kullanırsa bu durumda da edmonds karp algoritması olarak isimlendirilir.

Biz yazımızın konusu olan Edmonds Karp algoritmasını anlamak için sığ öncelikli olarak arama işlemini gerçekleştirelim:



BFS (sığ öncelikli arama) sonucunda bulunan yollarımız aşağıda listelenmiş olsun:

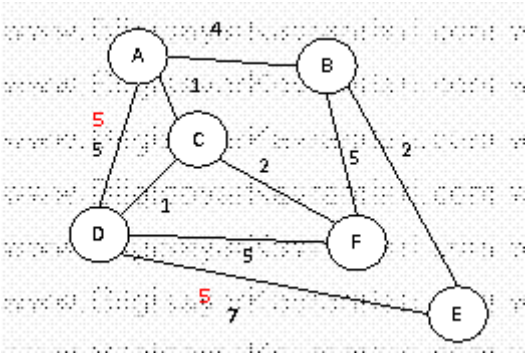
- A-D-E
- A-B-E
- A-C-D-E
- A-C-F-B-E
- A-C-F-D-E

- A-B-F-D-E

İlk bulacağımız yol (path) A-D-E olsun bu durumda algoritma bu yol üzerindeki en küçük kapasiteyi bulmaya çalışır:

min(A-D,D-E) = min(5,7) = 5

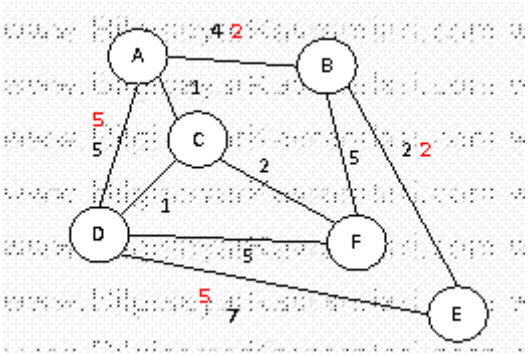
bu bulunan sayı aslında bu yol üzerinden akabilecek maksimum değerdir. Dolayısıyla bu yoldan 5 kapasitesinde akış yapılmasına karar verilir ve şekil üzerinde bu durum işaretlenir:



Yukarıdaki şekilde buluna kırmızı sayılar, o yoldaki anlık akış miktarlarıdır.

Ardından derin öncelikli arama işlemi devam eder ve örneğin A-B-E yolunu bulur.

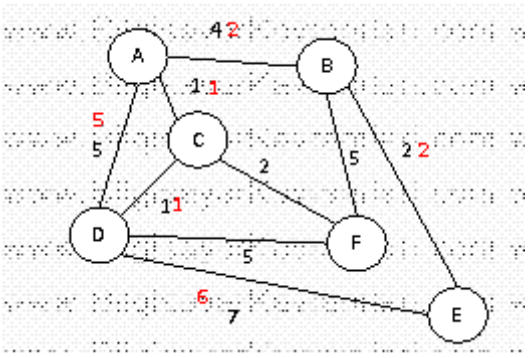
min(A-B, B-E) = min(4,2) = 2



Bir sonraki yolumuz olan A-C-D-E yolunu hesaplayalım.

min(A-C,C-D,D-E) = min(1,1,2) = 1

Yukarıda dikkat edilecek bir nokta, D-E aralığının şeklin ilk halinde olan 7 olarak alınması yerine 2 olarak alınmasıdır. Bunun sebebi şeklin bu kenarındaki 5 miktarındaki kapasitenin şu anda kullanılıyor olması ve artık geriye 2 miktarında kapasite kalmasıdır.



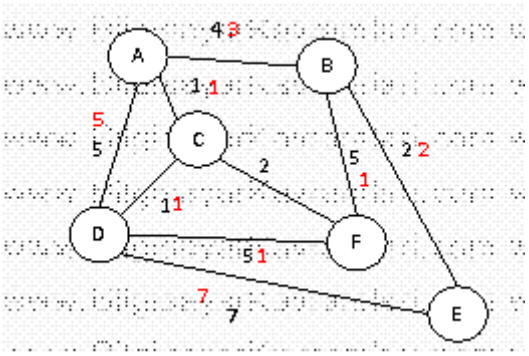
Bu adımsan sonra BFS algoritmamız ile bulduğumuz aşağıdaki yolları atlamamız gerekecektir:

- A-C-F-B-E
- A-C-F-D-E

Bunun sebebi bu yollarda bulunan A-C aralığının artık 0 kapasitesinin kalmış olması ve bu yollardan daha fazla akış elde edilememesidir.

Son olarak BFS algoritmamızın bulduğu A-B-F-D-E yolunu hesaplayalım:

min (A-B,B-F,F-D,D-E) = min (2,5,5,1) = 1

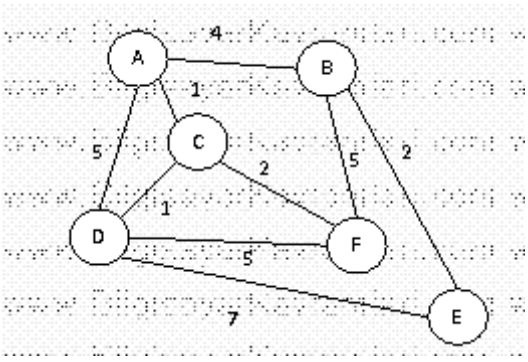


Yukarıdaki şekilde bütün düğümler arasındaki akışlar gösterilmiştir. Gerçekten de graftaki E düğümüne (hedef düğüme) gidebilen yollar maksimum kapasite ile doldurulmuştur.

SORU 13: Ford Fulkerson Algoritması

Bu algoritmanın amacı, literatürde azami akış (maximum flow) olarak geçen ve düğümler (nodes) arasında akış kapasiteleri belirli bir şekildeki (graph) bir başlangıçtan bir hedefe en fazla akışın sağlandığı problemleri çözmektir.

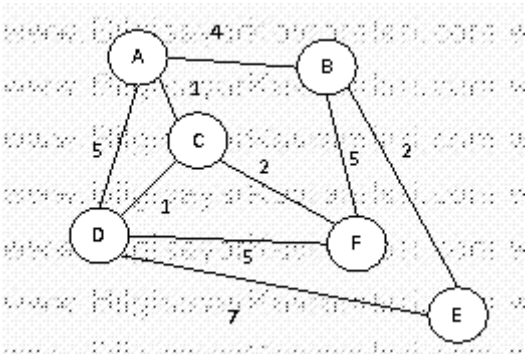
Azami akış (maximum flow) problemini örneğin şehirler arasında bağlı boru hattına veya tedarik zincirine benzetebiliriz. Örneğin aşağıdaki şekli ele alalım:



Buradaki düğümler, şehirleri ve düğümler arasındaki kenarlar (edges) ise şehirler arasındaki boru hatlarının kapasitesini belirtsin. Amacımız A düğümünden E düğüme azami miktarda akış sağlayabilmek olsun.

Ford-Fulkerson çözüm için yukarıdaki şekilde öncelikle hedef düğüme giden yolu bulur. Algoritma bu arama işlemi sırasında şayet derin öncelikli arama (depth first search ,DFS) kullanıyorsa ford fulkerson olarak isimlendirilir. Şayet aynı algoritma bu arama işlemi sırasında sığ öncelikli arama (breadth first search, BFS) kullanırsa bu durumda da edmonds karp algoritması olarak isimlendirilir.

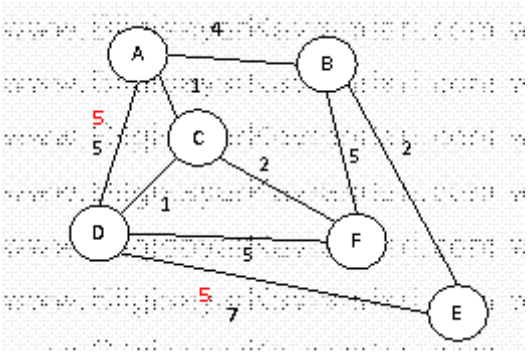
Biz yazımızın konusu olan ford fulkerson algoritmasını anlamak için derin öncelikli olarak arama işlemini gerçekleştirelim:



İlk bulacağımız yol (path) A-D-E olsun bu durumda algoritma bu yol üzerindeki en küçük kapasiteyi bulmaya çalışır:

min(A-D,D-E) = min(5,7) = 5

bu bulunan sayı aslında bu yol üzerinden akabilecek maksimum değerdir. Dolayısıyla bu yoldan 5 kapasitesinde akış yapılmasına karar verilir ve şekil üzerinde bu durum işaretlenir:

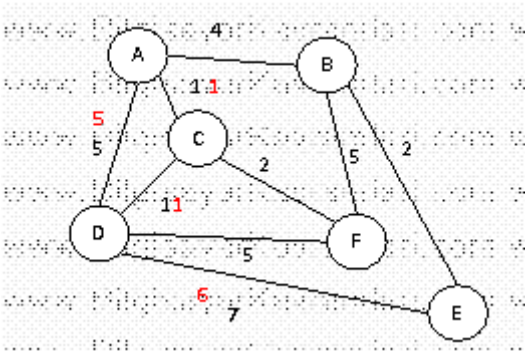


Yukarıdaki şekilde buluna kırmızı sayılar, o yoldaki anlık akış miktarlarıdır.

Ardından derin öncelikli arama işlemi devam eder ve örneğin A-C-D-E yolunu bulur.

min(A-C,C-D,D-E) = min(1,1,2) = 1

Yukarıda dikkat edilecek bir nokta, D-E aralığının şeklin ilk halinde olan 7 olarak alınması yerine 2 olarak alınmasıdır. Bunun sebebi şeklin bu kenarındaki 5 miktarındaki kapasitenin şu anda kullanılıyor olması ve artık geriye 2 miktarında kapasite kalmasıdır.



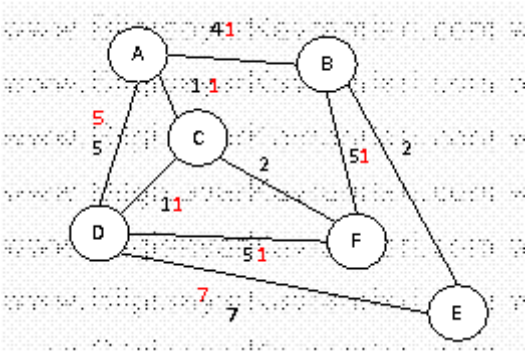
Bir sonraki derin öncelikli arama yolumuz A-C-F-D-E olsun. Bu durumda yol üzerindeki kapasite hesabı aşağıdaki şekilde olacaktır:

min(A-C,C-F,F-D,D-E) = min (0,2,5,7) =0

Görüldüğü üzere bu yol artık kullanılamaz çünkü bu yoldaki kapasite 0′dır. Bunun sebebi bir önceki adımda bulunan yolun, A-C arasındaki kapasiteyi sonuna kadar kullanmış olmasıdır.

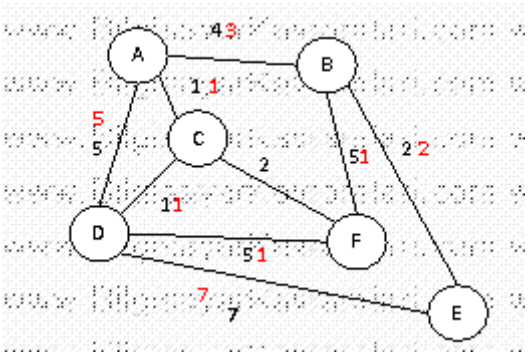
Sıradaki yolumuz A-B-F-D olsun:

min(A-B,B-F,F-D) = min(4,5,5,1) = 1



Son olarak yolumuz A-B-E olarak verilsin :

min(A-B,B-E) = min (3,2) = 2



Yukarıdaki şekilde bütün düğümler arasındaki akışlar gösterilmiştir. Gerçekten de graftaki E düğümüne (hedef düğüme) gidebilen yollar maksimum kapasite ile doldurulmuştur.

SORU 14: Şekillerde Sığ Öncelikli Arama

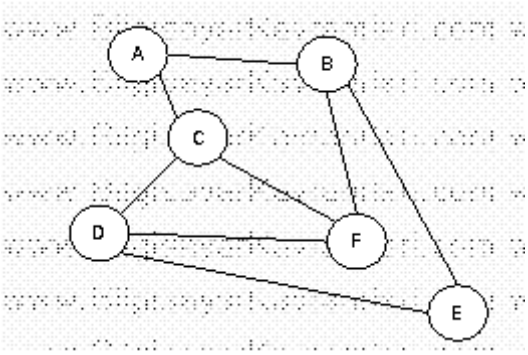
Bu yazının amacı genel amaçlı şekillerde (graphs) sığ öncelikli aramayı (breadth first search , BFS) açıklamaktır.

Bu yazı, ağaçlardaki sığ öncelikli arama ile karıştırılmamalıdır. Ağaçlar (trees) bilindiği üzere yönlü dairesel olmayan şekillerdir (directed acyclic graph) dolayısıyla ağaçlar üzerinde bu yazıda anlatılan sıra (queue) yapısına ihtiyaç duyulmaz.

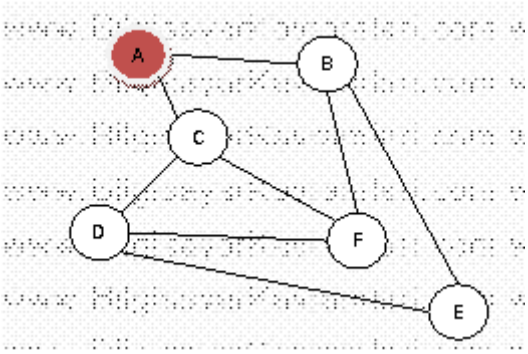
Sığ öncelikli arama bir başlangıç düğümünden başlayarak sıradaki komşuları dolaşan arama algoritmasıdır. Bu arama algoritmasında amaç önce başlangıç düğümüne yakın düğümlere bakmaktır.

Bu arama şekli suya atılan bir damlanın suda çıkardığı halkalara benzetilebilir. Başlangıç düğümüne en yakın halka sonra ikinci yakın halka ve sonra diğerleri şeklinde giden bir arama algoritmasıdır.

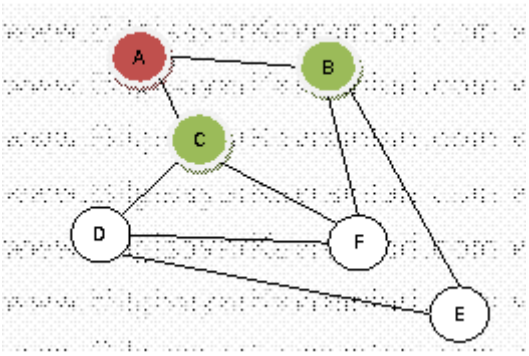
Bu durumu aşağıdaki örnek üzerinden anlamaya çalışalım:



Örnek olarak yukarıda bulunan şekli ele alalım. Örneğimizde başlangıç düğümü (node) olarak A düğümünü seçtiğimizi ve aranan düğümün E düğümü olduğunu düşünelim .

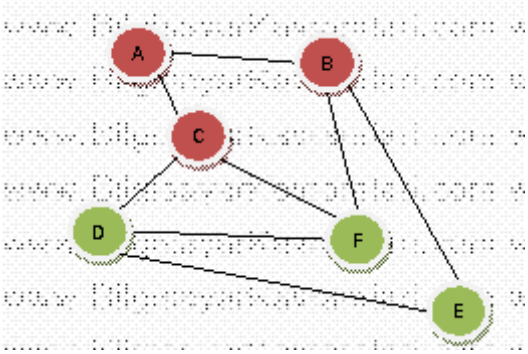


A düğümünün komşuluk listesini (adjacency list) çıkarıyoruz:



komşu(A) = {C,B}

Bu komşulara sırasıyla bakılıyor ve bu komşuların komşuluk listeleri çıkarılıyor:



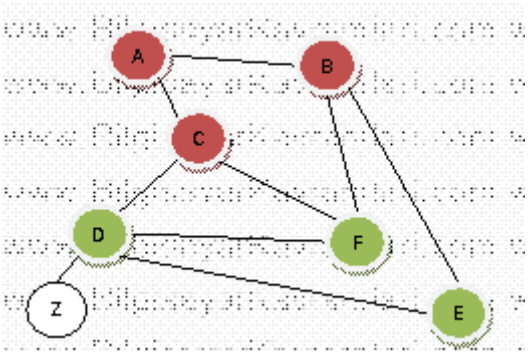
bakılanlar: {A,B,C}

komşular = komşu (B) , komşu (C) = {D,F,E}

Arana düğüm E, bu adımda bulunmuştur.

Yukarıdaki örnekte görüldüğü üzere bir sonraki derinliğe inilmeden önce, o derinlikte olan (başlangıç düğümüne o uzaklıkta olan) bütün düğümler dolaşılıyor ve ardından bir alt seviyeye iniliyor.

Yukarıdaki örnekte bütün ağaç dolaşılmış gibi düşünülebilir. Örneğin şekil (graph) aşağıdaki gibi olsaydı:



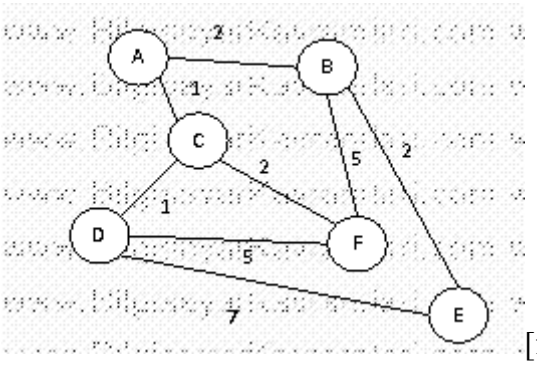
Bu durumda Z düğümüne hiçbir zaman bakılmayacaktı çünkü aranan düğüm, Z düğümünden daha sığ bir derinlikte olacaktı.

Kısacası BFS arama algoritması ile aranan düğümün derinliğine kadar olan bütün düğümlere bakılır ancak bu derinlikten daha aşağıda olan düğümlere bakılmasına gerek yoktur.

SORU 15: Dijkstra Algoritması

Bilgisayar bilimlerinde kullanılan ve algoritmayı literatüre kazandıran kişinin ismini taşıyan dijkstra algoritması, verilen bir şekilde (graph) en kısa yolu (shortest path) bulmak için kullanılır.

Bu algoritmanın çalışmasını örnek bir şekil(graph) üzerinden göstermeye çalışalım. Bu gösterimin ardından Dijkstra algoritmasının başarısını ve zafiyetlerini tartışabiliriz.

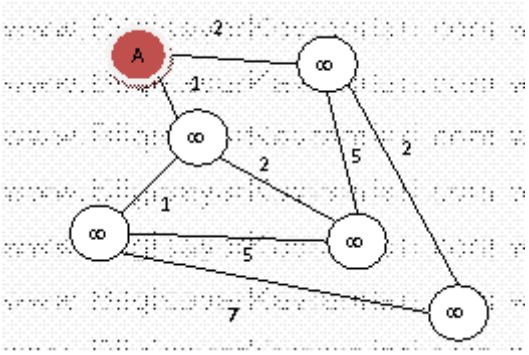


[flashvideo file=http://www.bilgisayarkavramlari.com/wp-content/uploads/dijkstra.flv /]

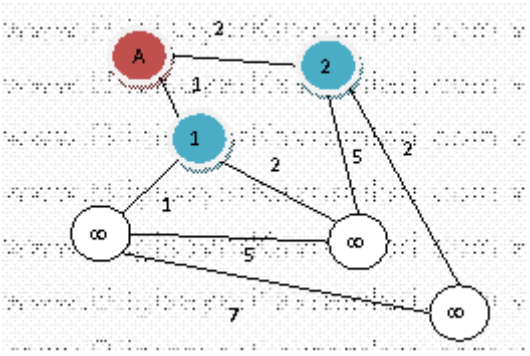
Örnek olarak yukarıda bulunan şekli ele alalım. Dijkstra algoritması herhangi bir şekildeki bir düğümden diğer bütün düğümlere giden en kısa yolu hesaplar.

Örneğimizde başlangıç düğümü (node) olarak A düğümünü seçtiğimizi düşünelim ve algoritmanın çalışmasını bu düğümden başlayarak gösterelim.

Algoritma başlangıçta bütün düğümlere henüz erişim olmadığını kabul ederek sonsuz (👈) değeri atar. Yani başlangıç durumunda henüz hiçbir düğüme gidemiyoruz.

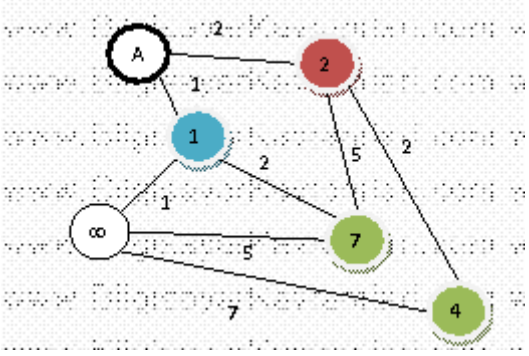


Ardından başlangıç düğümünün komşusu olan bütün düğümleri dolaşarak bu düğümlere ulaşım mesafesini günceller.



Bu güncelleme işleminden sonra güncellenen düğümlerin komşularını günceller ve bütün düğümler güncellenene ve şekil üzerinde yeni bir güncelleme olmayana kadar bu işlem devam eder.

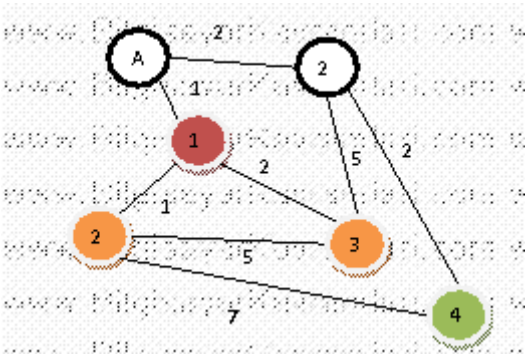
Örneğin yukarıda A düğümünün komşusu olan düğümler (B ve C) güncellendiler. Bir sonraki adımda bu düğümlerin komşuları güncellenecektir. İstediğimiz bir düğüm ile başlayalım. Örneğin önce B sonra C düğümünün komşularını güncelleyelim:



Yukarıdaki şekilde, B düğümünün komşusu olan E ve F düğümleri güncellenmiştir. Bu güncelleme işlemi sırasında B düğümünün üzerindeki mevcut maliyet ile E ve F düğümlerine gidişten doğan maliyet toplanmıştır.

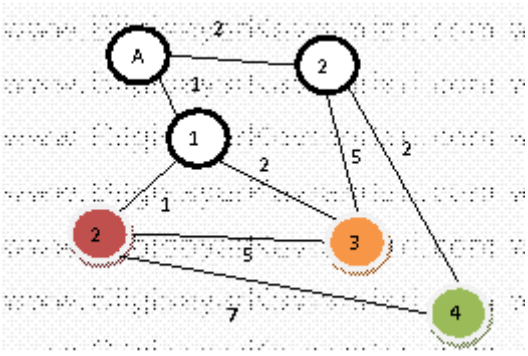
Örneğin E düğümü için 2 + 5 = 7 ve F düğümü için 2 + 2 = 4 değerleri bulunmuştur.

Şimdi bir sonraki düğüm olan C düğümünden güncelleme yapalım:



Bu aşamada güncellenen düğümler, C düğümünün komşusu olan D ve F düğümleridir. Bu düğümlerden, F düğümü daha önce güncellenmişti ve değer olarak 7 taşımaktaydı, ancak yeni gelen değer, F için C üzerinden 1 + 2 = 3 olarak hesaplanmış ve bu değer daha önceden hesaplanan 7 değerinden düşük olduğu için 7′nin üzerine yazılarak 3 olmuştur.

Algoritmanın çalışmasını sonraki düğümlerin güncellenmesi ile sürdürelim. Bu sefer örneğin D düğümünün komşularını güncelleyelim (şu anda D, F ve E düğümleri aynı derecede yeni güncellenmiştir bunlardan herhangi birisi ile başlanması Dijkstra algoritmasına uygundur. Ancak recursive (özyineli) bir kodlama yapılıyorsa koda göre elbette yakın olan komşular güncellenecektir)

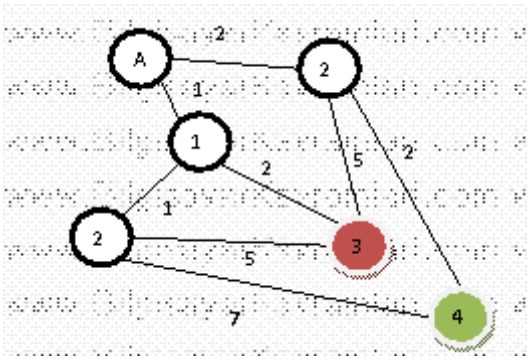


D düğümünün hiçbir komşusu güncellenmemiştir bunun sebebi hesaplanan değerlerin mevcut değerlerden yüksek olması ve dolayısıyla daha iyi bir sonuç bulunamamasıdır.

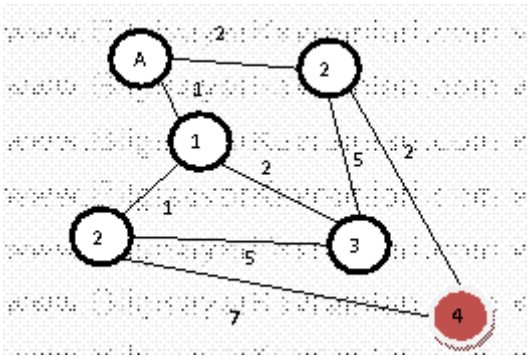
F için 2 + 5 = 7 > 3

E için 2 + 7 = 9 > 4

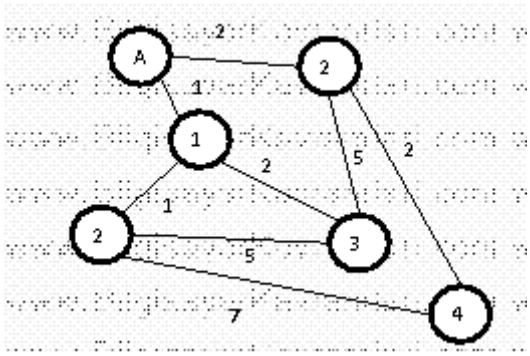
Dolayısıyla bir sonraki düğümü seçerek devam ediyoruz



F düğümünün komşularında da bir değişiklik olmuyor ve son olarak E düğümü deneniyor:



Son halimizde de bir değişiklik olmayarak şeklimiz (graph) kararlı bir halde (daha fazla değişiklik olmadığı için) dolaşılmış ve bitmiş oluyor.

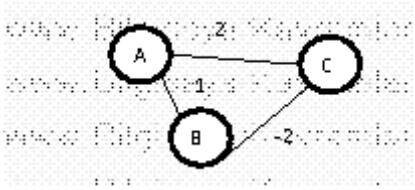


Yukarıdaki bu grafta, düğümler üzerinde yazılı olan değerler, A düğümünden başlanarak her düğüme gidilebilen en kısa yol mesafesini vermektedir. Örneğin şekildeki F düğümüne ulaşma maliyeti 3 olarak hesaplanmıştır ve Dijkstra algoritması, A düğümünden F düğümüne daha kısa bir yol bulunamayacağını iddia eder (eşit farklı yollar bulunabilir ama en kısa yol yine de 3 olur)

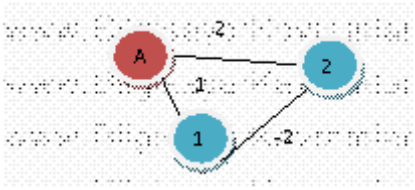
Dijkstra algoritmasının zayıf yönü

Algoritma ne yazık ki eksi (-) değer taşıyan bir kenar bulunması halinde başarılı çalışmaz. Bunun sebebi eksi (-) değerdeki kenarın sürekli olarak mevcut durumdan daha iyi bir sonuç üretmesi ve algoritmanın hiçbir zaman için kararlı hale gelememesidir.

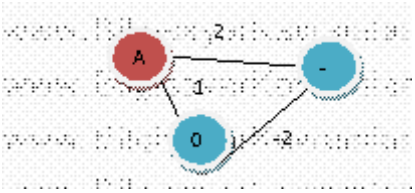
Örneğin aşağıdaki basit şekli ele alalım:



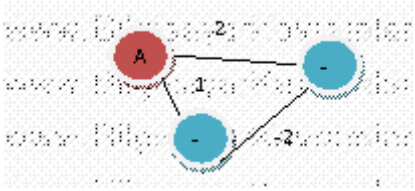
Yukarıdaki bu şekilde B-C kenarı (edge) -2 değerinde verilmiştir. Bu durumda A düğümünden başlayarak en kısa yolları hesaplamak istersek, öncelikli olarak A düğümünden komşusu olan düğümlere güncelleme yapılacaktır:



Ardından bu düğümlerin komşuları güncellenecektir. İki düğümde birbirine -2 değerini ekleyerek daha kısa bir yol bulacak ve birbirlerini güncelleyerek daha kısa sonuçlara ulaşacaktır.



Bu işlem ne yazık ki sonsuza kadar giden bir sürecin başlangıcıdır ve hiçbir zaman daha iyi bir sonuç bulma işlemi bitmez.



Sürekli olarak mevcut durumdan daha iyi bir durum bulunacak ve daha küçük değerler güncellenecektir. Bu yüzden Dijkstra algoritması, eksi (-) değer taşıyan şekillerde (graph) başarılı çalışmaz.

SORU 16: Hasse Çizgeleri (Hasse Diagrams)

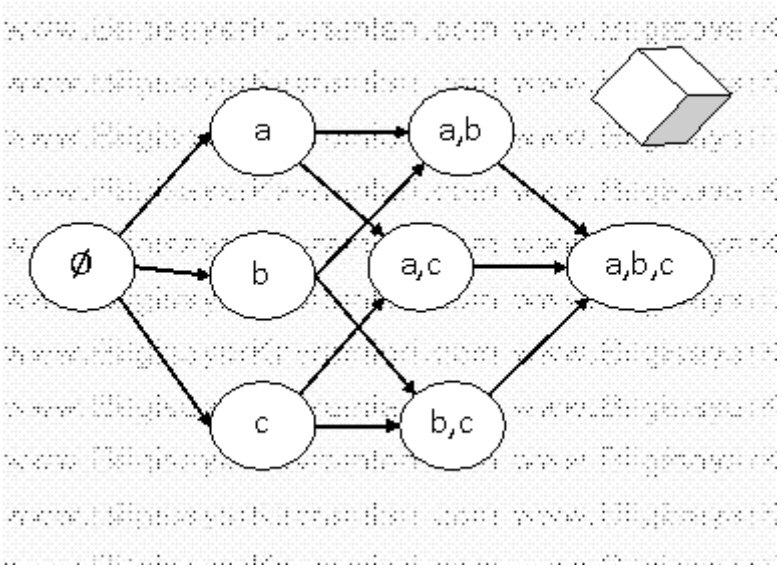
Bilgisayar bilimleri de dahil olmak üzere çok sayıdaki bilim ve mühendislik alanında kullanılan bir modelleme biçimidir. Şekilde (graph) kullanılan düğümler (nodes) birer kümeyi ifade etmektedir. Çizimdeki geçişler (transitions) bir kümeden diğer kümeye bir eleman ile geçilebilme durumunu ifade eder. Buna göre bir kümeye eleman eklenmesi veya eleman çıkarılması bir adımlık bir iştir ve her iş bir geçiş olarak değerlendirilmelidir.

Konuyu örnekler üzerinden anlamaya çalışalım. Örneğin kümemiz {a,b,c} olsun ve bu kümenin alt kümelerini hasse diyagramı (hasse diagram) olarak çizelim.

Öncelikle kümemizin alt kümelerinin kümesini (güç kümesini, power set) bulalım:

{Ø , {a}, {b}, {c}, {a, b}, {b, c} , { a,c} ,{a,b,c}}

Bu kümenin elemanları arasındaki hasse diyagramını çizelim:



Yukarıda görüldüğü üzere boş kümeden başlayarak her küme eklentisi ayrı bir okla gösterilmiştir. Yukarıdaki şekil, 3 boyutlu bir küpe bu açıdan benzetilebilir (yanda temsili bir küp çizilmiştir). Yukarıdaki şekilde, kümeye eleman eklenmesi bir geçiş durumu olarak tanımlanmış ve her geçiş, kümeye tek bir eleman eklenmesi olarak gösterilmiştir. Örneğin boş kümeden kümenin tamamı olan {a,b,c} kümesine geçiş mümkündür ama bu dolaylı geçiş (transition) doğrudan bir bağlantı ile gösterilmek yerine olduğu gibi yani dolaylı bir şekilde bırakılmıştır. Dolayısıyla şekildeki mesafe, bir kümeden diğer kümeye olan geçiş için gerekli adım sayısını vermektedir. Örneğin {a} kümesinden {a,b,c} kümesine geçiş için iki farklı yol bulunur ve ikisinde de iki adımda geçiş tamamlanır (ya {a,b} ya da {a,c} üzerinden)

SORU 17: Kırmızı-Siyah Ağaçları (Red Black Trees)

Bilgisayar bilimlerinde, veriyi ağaçta (tree) tutarken, ağacın dengeli (balanced) olmasını sağlayan bir algoritmadır. Algoritma, veriyi tutuş şekli sayesinde, arama, ekleme veya silme gibi temel işlemlerin en kötü durum analizi (worst case analysis) O(logn)'dir, yani algoritma n elman için bu işlemleri en kötü O(logn) zamanda yapmaktadır.

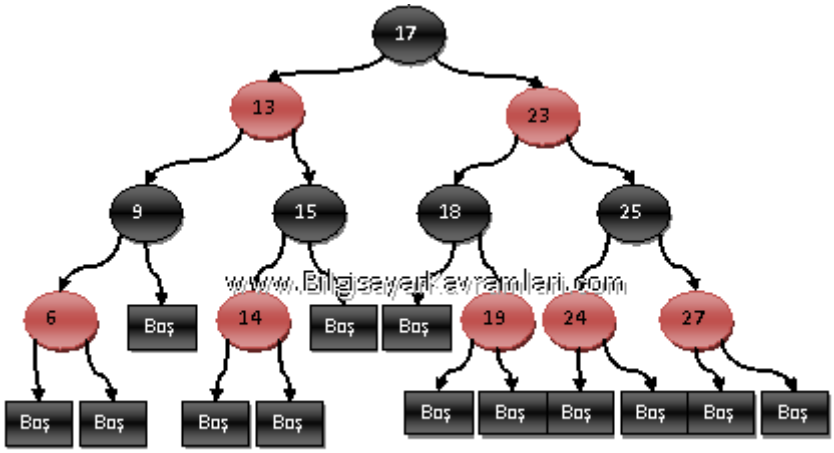
Kırmızı-siyah ağaçlar (red-black trees) tanım itibariyle ikili arama ağaçlarıdır (binary search tree) ve bu anlamda, herhangi bir düğümün solunda kendisinden küçük ve sağında ise büyük verilerin durması beklenir. Ağaçta ayrıca her düğüm için bir renk özelliği tutulur. Yani bir düğüm kırmızı veya siyah renk özelliği taşıyabilir. Ağaçtaki düğümlerin taşıması gereken bu özellikler aşağıdaki şekilde sıralanabilir:

1. Ağaçtaki her düğüm kırmızı ya da siyahtır
2. Kök düğüm (root node) her zaman için siyahtır.
3. Bütün yaprak düğümler (leaf nodes) siyahtır
4. Herhangi bir kırmızı düğümün bütün çocukları siyahtır.

5. Herhangi bir düğümden, yaprak düğüme kadar gidilen bütün yollarda eşit sayıda siyah düğüm bulunur.

Yukarıdaki bu kurallar ışığında, herhangi bir düğümden, yapraklara kadar olan yolun, gidilebilecek en kısa yolun iki mislinden kısa olduğu garanti edilebilir. Diğer bir deyişle, ağacın aynı seviyedeki düğümleri aynı renktir. Ayrıca ağaçtaki renklendirme kökten başlayarak, siyah – kırmızı – siyah – kırmızı sıralamasıyla değişmektedir.

Örneğin aşağıdaki ağacı ele alalım:



Yukarıda görülen örnek ağaçta, kök düğüm siyah, yapraktaki boş düğümler (null) siyah ve bu düğümler dışındaki her düğümün çocukları, kendisinin ters rengindedir. Örneğin kırmızı olan 13 düğümünün çocukları siyah, siyah olan 25 düğümünün çocukları ise kırmızıdır. Bu anlamda kutu ile gösterilen yaprak düğümler göz ardı edilirse, aynı seviyedeki düğümler aynı renkte olmaktadır.

Arama işlemi (search)

Ağaç üzerinde bir değerin aranması, ikili arama ağacında (binary search tree) gibi yapılır. Yani aranan değer önce kök düğümde (root node) aranır, şayet aranan değer daha büyükse sağa, küçükse sola devam edilir. Nihayetinde aranan değer bulunana veya boş (null) değere ulaşılana kadar bu işlem devam eder.

Ekleme işlemi (insertion)

Ağaca bir düğümün nasıl eklendiğini algoritmasal olarak aşağıdaki şekilde açıklayabiliriz:

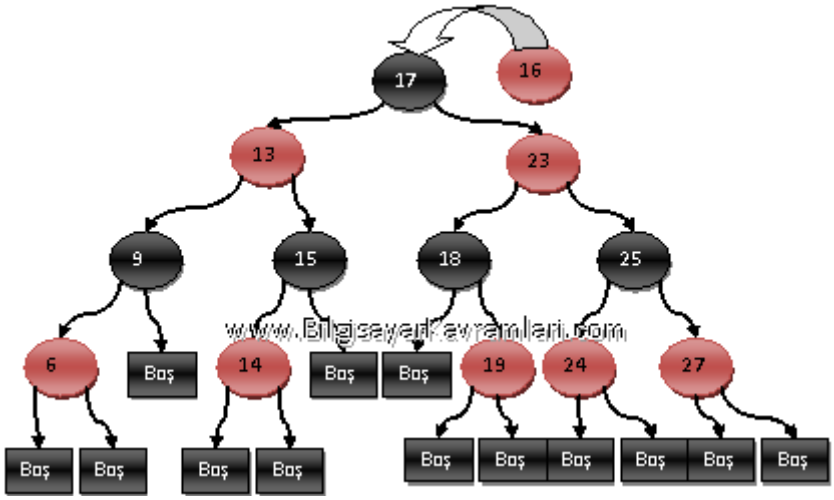
Ekleme işlemi normal bir ikili arama ağacına (binary search tree) ekler gibi başlar. Bu işlem sırasında yeni düğümün kırmızı olacağını kabul ederiz.

Ekleme işlemi sırasında uyulması gereken 3 temel kural vardır:

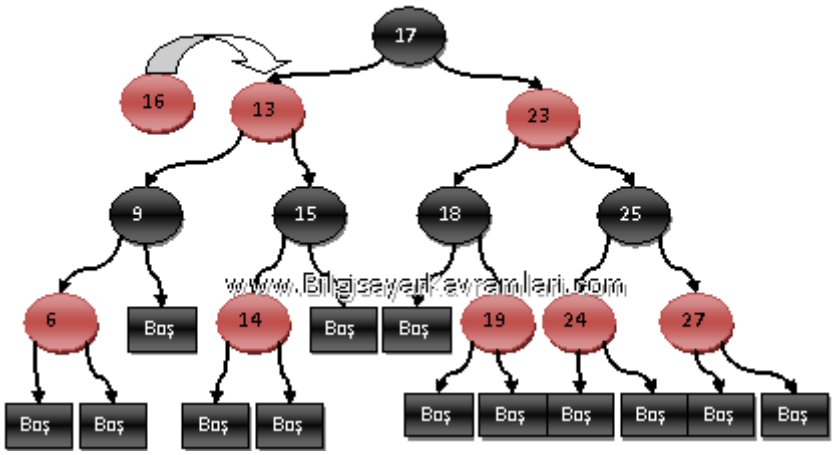
- Kök düğüm (root node) her zaman için siyahtır.
- Herhangi bir düğümden, yapraklara kadar uzanan herhangi bir yolda, eşit sayıda siyah düğüm bulunur.

- Bir kırmızı düğümün, kırmızı çocuğu bulunamaz.

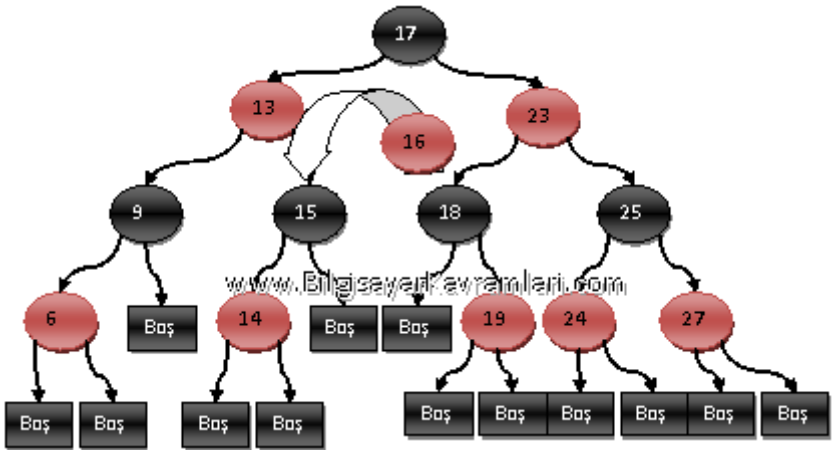
Yukarıdaki istenmeyen durumlardan birisi oluştuğunda, ağaçtaki düğümlerin rengi değiştirilir ya da değiştirilemiyorsa ağaçta dengelemek için döndürme (rotation) işlemi yapılır.



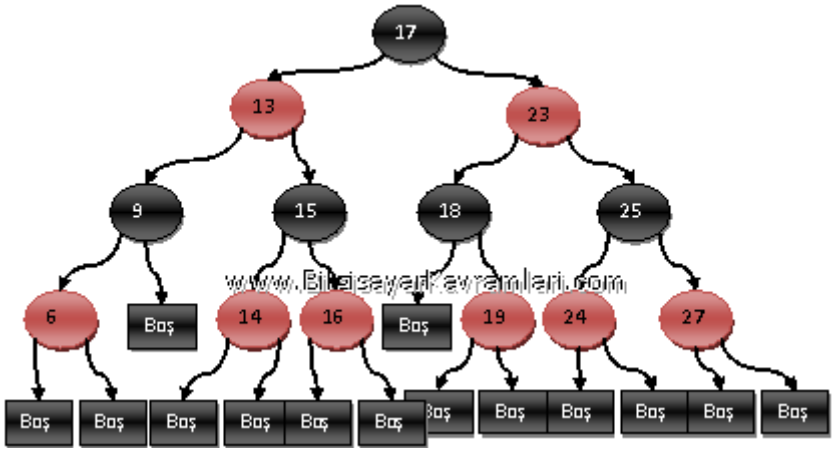
Örneğin, yukarıdaki şekilde gösterildiği üzere 16 sayısını ağaca eklemek isteyelim.



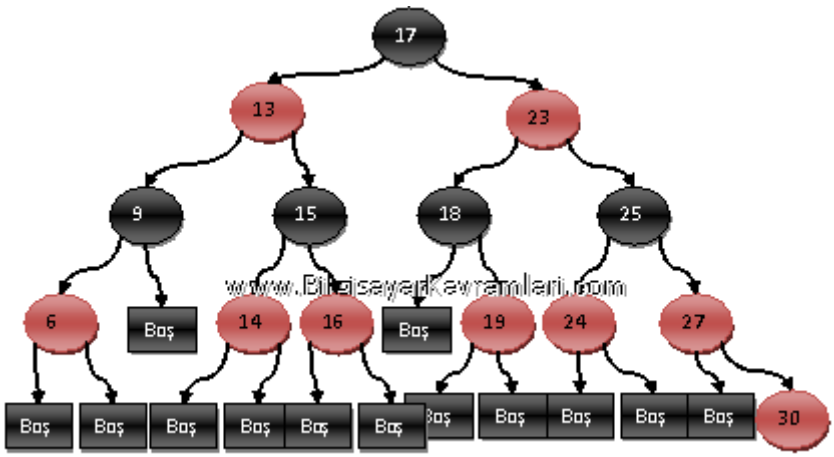
16 değeri, 17′den küçük olduğu için, klasik bir ikili arama ağacında hareket eder gibi , ağacın sol tarafında devam edilecek ve 13 ile karşılaştırılacak.



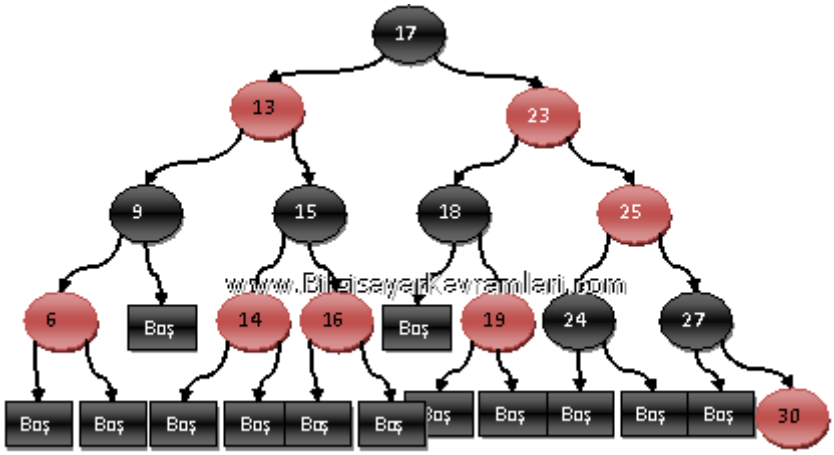
13′ten büyük olduğu için sağa bakılıyor. Ve 15′ten de büyük olduğu için 15′in sağına ekleniyor.



Görüldüğü üzere yeni eklenen 16, 15′ten büyük ve sağındaki boş yere yerleştirilmiştir. Bu yerleştirme sonucunda iki ardışık kırmızı durumu oluşmadığı için sorun yoktur. Yani bir siyah düğüm altına kırmızı düğüm eklenmiştir. Şimdi örnek olarak sırasıyla 30 ve 32 sayılarını ekleyelim.

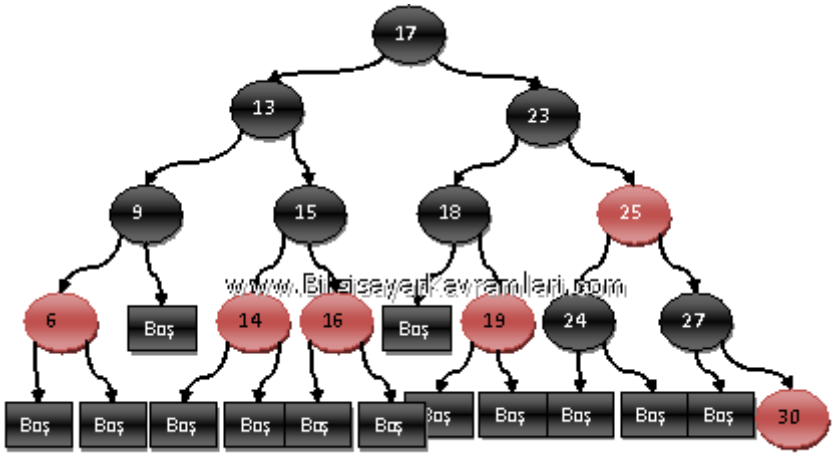


Ağaca yukarıdaki şekilde 30 değeri eklenince, ağacın en sağına yerleşmekte ve istenmeyen bir durum olan iki kırmızı arka arkaya gelmektedir. Çözüm olarak ağaçtaki düğümlerin rengi değiştirilmelidir.

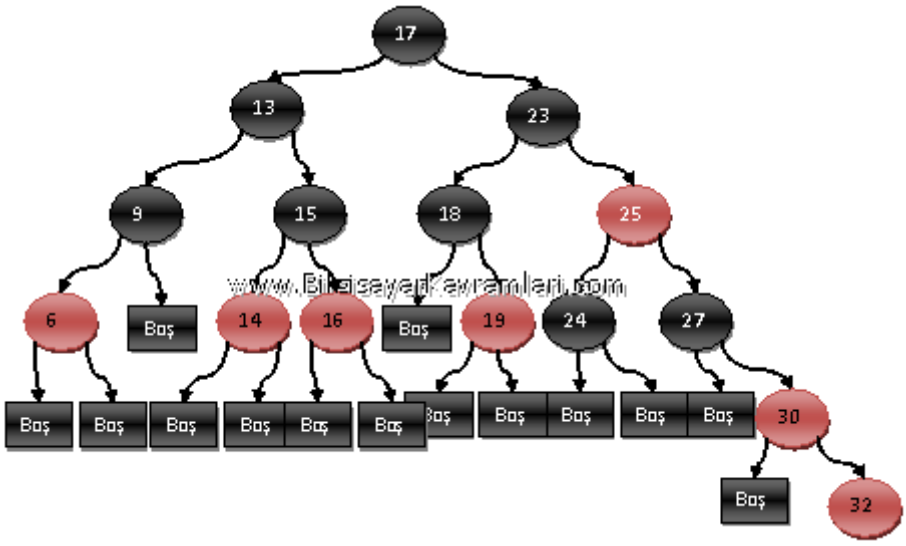


Öncelikle, 30'un hemen üzerinde bulunan 27 düğümü, iki kırmızı düğüm arka arkaya olamayacağı için siyaha çevrilir. Siyaha çevrilen 27 numaralı düğümün büyük babası 23 numaralı düğümdür. Dolayısıyla 27 numaralı düğümün amcası 18 numaralı düğüm olur. 25 numaralı düğümün kırmızıya çevrilmesi, 23 numaralı büyük baba için problem oluşturur çünkü bir kırmızı düğümün çocukları siyah olmalıdır. Aynı zamanda problem 17 numaralı düğümden yaprağa kadar giden yolda da eşit miktarda siyah düğüm bulunmalıdır. Yukarıdaki örnekte 23 numaralı düğümün siyah kalması durumunda, 17 numaralı düğümden gidilebilen sağ yol ile sol yol arasında düğüm sayıları farklı olmaktadır.

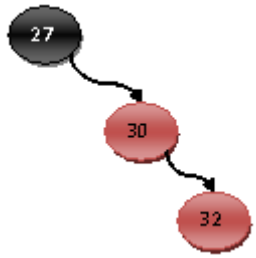
Çözüm olarak 17 numaralı düğüm kırmızı yapılabilir ancak bu durumda da kök düğümün kırmızı olamayacağı kuralı ile ihtilafa düşülür. Dolayısıyla bu adımda çözüm 13 ve 23 numaralı düğümlerin siyah yapılmasıdır.



Ağacın çalışma şeklini daha iyi anlayabilmek için ağaca bir de 32 değerinde bir düğüm eklemeyi deneyelim:

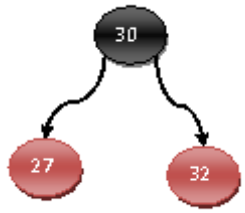


Yukarıdaki örnekte görüldüğü üzere, 32 değeri, bir önceki adımda eklediğimi 30 değerinin sağına gelmektedir. Hemen dikkat edilebilecek bir problem, 30-32 ikilisinin arka arkaya kırmızı olmasıdır. Bu durumda 30 düğümünün siyah yağılması veya 32 düğümünün siyah yapılması problemi çözmez çünkü kökten yapraklara kadar giden yolda eşit sayıda siyah düğüm bulunmalıdır. Bu yola yeni bir siyah düğüm eklenmesi bu dengeyi bozar. Çözüm olarak saat yönünün tersine döndürme işlemi uygulanıp çocukların siyah yapılması gerekir. Bu durumu aşağıdaki şekil üzerinden açıklamaya çalışalım:



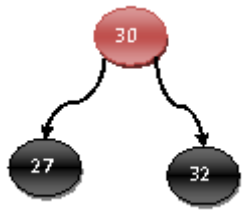
Yukarıdaki 2 problem bulunuyor:

1. İki kırmızı düğüm arka arkaya
2. Yolda tek siyah düğüm bulunmasına izin verilmiş, yapraklara kadar giden ikinci bir siyah düğüm çıkarma hakkımız yok.

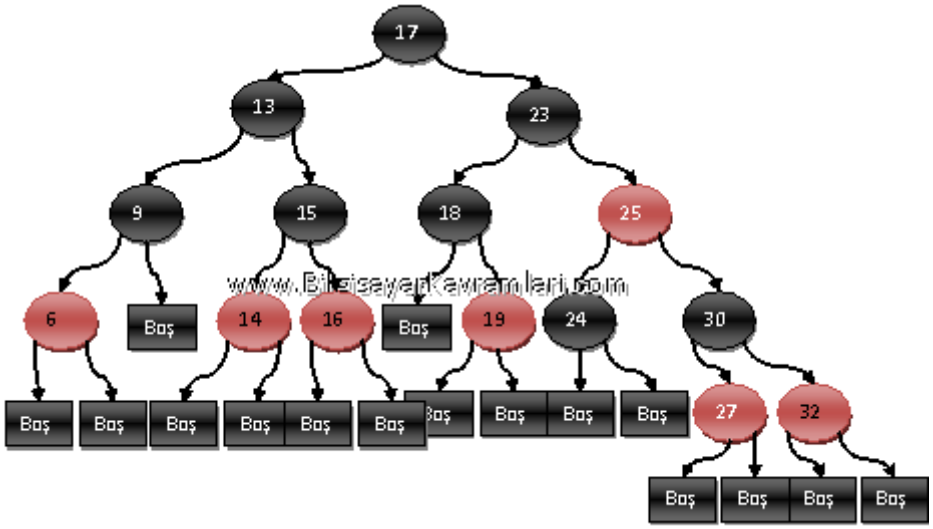


Yukarıdaki yeni halinde, kural bozulmaksızın hem tek siyah ile yaprağa ulaşılıyor hem de iki kırmızı ihlalinden kurtulunuyor.

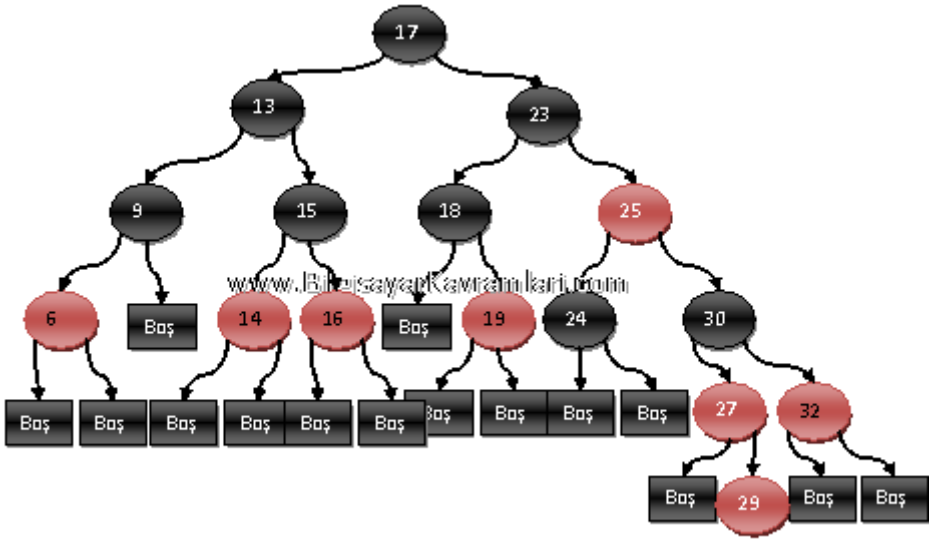
Benzer bir çözüm aşağıdaki şekilde olabilirdi:



Ancak yukarıdaki bu yeni halimiz ağacın bütünüyle uyuşmamaktadır. Yani döndürme işleminin yapıldığı ağacın üst düğümü 25 olduğu ve kırmızı olduğu için bu ikinci çözüm yeni bir kırmızı kırmızı komşuluğu doğuracaktır. Dolayısıyla ilk örnekte olduğu gibi problemi çözebiliriz:

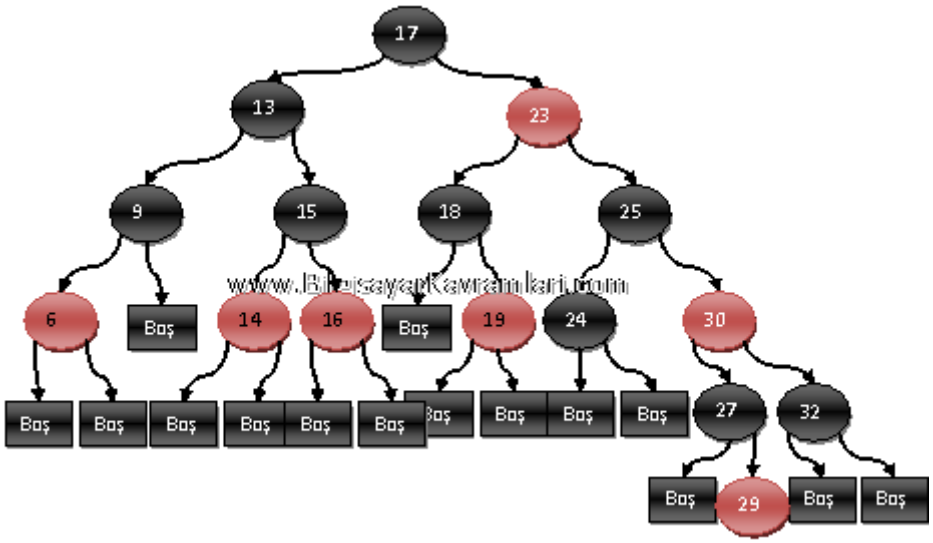


Yukarıdaki yeni halimizde kuralların tamamına uyarak ağaca yeni sayıyı eklemiş bulunuyoruz. Benzer bir durumu oluşturup daha yukarıdan döndürme (rotation) işleminin nasıl yapıldığını, ağaca 29 sayısını ekleyerek görebiliriz. Bakın 29 sayısı eklenince de yapılan işlem aslında aynıdır sadece daha büyük bir döndürme işlemi gerçekleştirilmiş olacaktır:



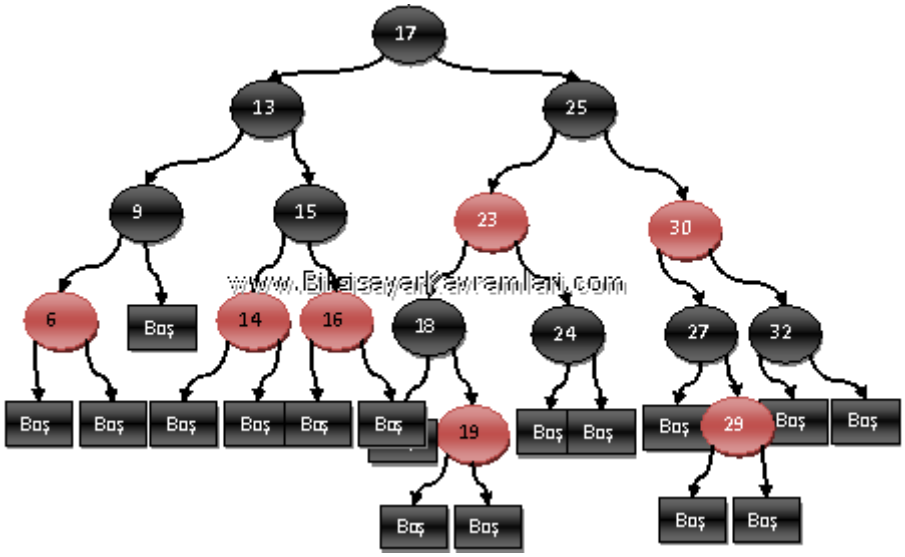
İki kırmızı durumu oluşuyor. Bu noktada artık ikisinden birisinin siyah olması sorunu çözmez çünkü yaprağa kadar olan yolda her halükarda bir fazla siyah düğüm oluşur.

Çözüm olarak sırasıyla kırmızı-siyah dönüşümü yapıyoruz:



Görüldüğü üzere köke kadar giden yolda kırmızı-kırmızı durumunu engellemek için düğümler bir kırmızı bir de siyah sırasına dönüştürüldü. Şimdi problem 23 numaralı düğümün sağ ve sol taraflarındaki yol uzunluğunun farklı olması. Yani 23 numaralı düğümün sol tarafındaki problem çözülürse 17 numaralı düğüm için bir problem yok

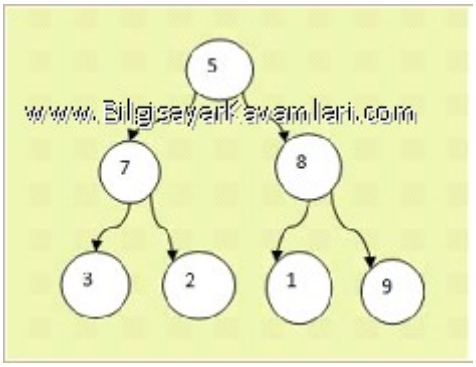
Döndürme işlemi gerçekleştiriyoruz:



Yukarıda görüldüğü üzere 25 numaralı düğüm23 ve 30 numaralı düğümler arasındayken, bu düğümlerin atası durumuna geçmiştir.

SORU 18: Yerleşim Sıralaması (Topological Sort, İlinge Sıralaması)

Bilgisayar bilimlerinde, yönlü dairesel olmayan şekiller (directed acyclic graphs) üzerinde çalışan bir sıralama algoritmasıdır (sorting algorithm). Algoritma tanımı itibariyle tek köklü ve çok köklü ağaçlar (tree) üzerinde çalışmak için tasarlanmıştır denilebilir. Örneğin aşağıdaki ağacı ve sıralama algoritmasının çalışması sonucunda elde edilen neticeyi inceleyerek konuyu anlamaya başlayalım:



Yukarıdaki şeklin, yerleşim sıralaması aşağıdakilerin herhangi birisi olabilir:

5783219

5782319

5781291

5873219

5872319

5871291

5732819

5723819

5732891

5723891

5819732

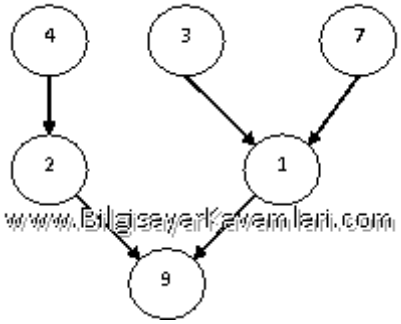
5891732

5819723

5891723

Görüldüğü üzere yerleşim sıralamasındaki amaç, bir şekilde daha üst seviyede bulunan, veya bir düğümü göstermekte olan diğer düğümün, sıralama sonucunda daha önce gelmesidir. Yukarıdaki şekilde örneğin 5 numaralı düğüm, 7 ve 8 numaralı düğümleri işaret etmektedir. Bu durumda, sıralama sonucunda hiçbir şekilde 7 veya 8, 5′ten önce gelemez.

Yukarıdaki örnek, tek köklü bir ağaçtır ve bu ağaçta sıralama sonucunda her zaman için kök düğüm (yukarıdaki örnekte 5 numaralı düğüm) sıralama sonucunun başında yer almalıdır. Ancak yerleşim sıralaması algoritması (toplogy sort algorithm) birden fazla kökü bulunan şekiller için de çalışmaktadır. Bu durumda, kök olan bütün düğümler aynı seviyede kabul edilir. Örneğin aşağıdaki şekli ele alalım:



Yukarıda verilen yerleşimin sıralanma alternatifleri aşağıda listelenmiştir:

437219

473219

347219

374219

734219

743219

437129

473129

347129

374129

734129

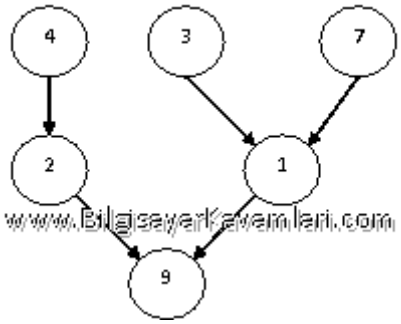
743129

…

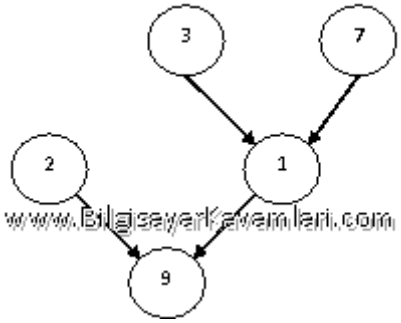
Yukarıdaki sonuçların elde edilmesini sağlayan algoritmayı aşağıdaki şekilde tanımlayabiliriz:

```
A: Boş bir liste olsun (veya dizi)
Şekil üzerinde (graph) kendisine gidilmeyen bir düğüm varken
   Bu düğümü alıp A listesine ekle
   düğümü şekilden (graph) sil
```

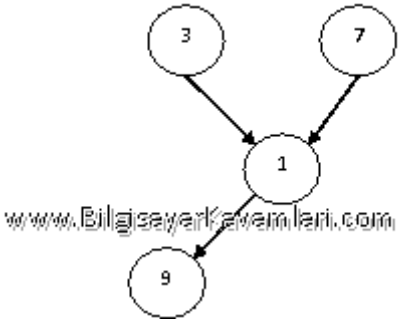
Görüldüğü üzere sıralama algoritmasında kendisine gidilmeyen düğümlerden herhangi birisini (ki bu düğüm aslında köklerden birisidir) alarak ilerliyoruz. Bu düğüm silindikten sonra yeni kökler çıkıyor. Örneğin yukarıdaki şekil sıralanırken aşağıdaki adımlardan geçebilir:



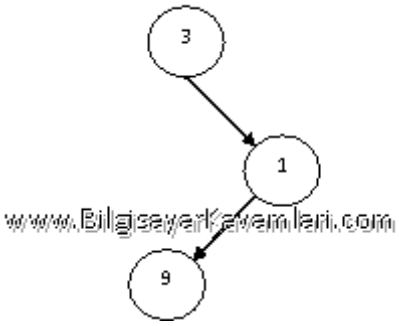
Sırlama için örneğin 4′ü alalım. Sonuç : 4



Bu düğüm silindi artık yeni köklerimiz : 2,3,7 oldu ve biz örneğin 2′yi alalım. Sonuç : 42



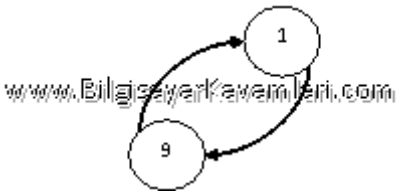
Yeni halinde alabileceğimiz kök düğümler sadece 3 ve 7′dir. Örnek olarak 7′yi alalım. Sonuç 427



Yeni halinde sadece 3 alınabilir çünkü diğer düğümlerin hepsine (1 ve 9) bir şekilde gidilen bir başka düğüm bulunmaktadır. Sonuç 4273

Son olarak sırasıyla 1 ve sonrada 9′u alırsak sıralama işlemimiz tamamlanmış olur: Sonuç 427319 olarak bulunur.

Yukarıdaki algoritmanın, daire içeren bir şekilde (graph) çalışmayacağı barizdir. Bunun sebebi daireye ulaşıldığında bir şekilde kendisine gitmeyen bir düğüm elde edilememesidir. Örneğin graf (graph) aşağıdaki şekilde verilmiş olsun:



Yukarıdaki durumda, ne 9 ne de 1 alınamaz çünkü ikisini de gösteren birer düğüm bulunmaktadır. Bu durumda algoritmamız çalışmaz, zaten yerleşim sıralamasına göre (topological sort) hangi düğümün önce geleceği de belirsizdir.

SORU 19: Yansıma(Reflexivity)

Yansıma, bilgisayar bilimlerinin de içerisinde bulunduğu bir grup bilim ve mühendislik alanında kullanılan mantıksa gösterimler ve muntazam diller (Formal languages) ve ayrıca bu dillerin dayandığı matematik ve mantıksal gösterimler sırasında kullanılan temel özelliktir.

Yansıma özelliği bir bağıntı (relation), matris ya da şekilde (graph) üzerinde bir işlemin geri gelebilmesi, yansıyabilmesi veya ters dönebilmesi olarak düşünülebilir. Örneğin A matrisi için x ∈A, (x,x) ∈ R yazılabilir. Veya bir A bağlantı kümesi için x ∈ A, x R x.yazmak doğrudur. Bu anlatımı görsel olarak göstermemiz gerekirse:

	1	2	3	4
1	■			
2		■		
3			■	
4				■

Yukarıdaki matriste görülen köşegen (diyagon) değerleri tanımlı olan kare matrise yansıma özelliği olan matris ismi verilir.

Benzer şekilde bir bağıntıda (relation) bulunan bütün elemanların kendi üzerinde yansıması tanımlı olmalıdır. Örneğin aşağıda verilen bağıntılardan hangilerinin yansıma özelliği (reflexivity) olduğunu bulmaya çalışalım.

R_1={ (1,1),(2,2),(3,3),(4,4) }

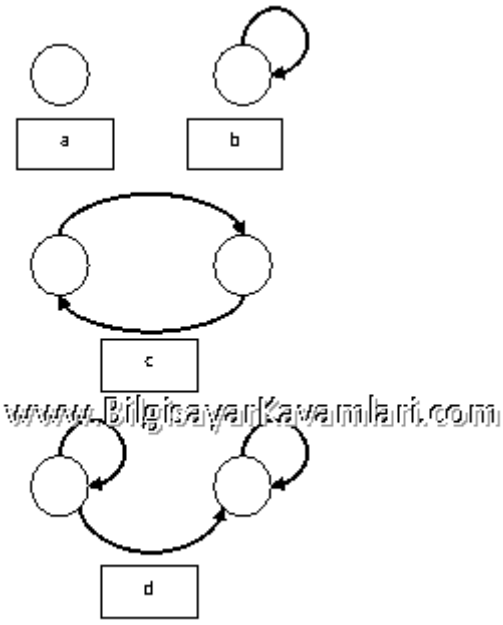
R_2={ (1,2),(2,3),(3,2),(2,1) }

R_3={ (1,1),(2,2),(2,1),(1,2) }

R_4={ (3,1),(2,2),(3,3),(1,1) }

Yukarıda verilen bağıntılardan (relation) sadece R_2yansıma özelliği olmayan bir bağlantıdır. Bunun dışındaki R_1, R_4 ve R_3için yansıma özelliği vardır diyebiliriz. Sebebi bu bağlantılarda bulunan bütün değerlerin kendi üzerinde yansıması mümkündür. Örneğin R_1için kullanılan 1,2,3 ve 4 değerleri veya R_2için kullanılan 1 ve 2 değerleri bu bağlantıların kendi içlerinde yansımışlardır (reflexive)

Yukarıda matris (masfuf) ve bağlantı için açıkladığımız yansıma özelliğini şekillere (graph) uygulamamamız da mümkündür. Örneğin aşağıdaki şekilleri ele alalım:



Yukarıdaki şekillerden sadece b ve d şekillerinin yansıma özelliği bulunur. Bunun sebebi bu şekillerde bulunan düğümlerin (nodes) tamamının kendisine dönebilmesidir. Buna karşılık c şeklinde veya a şeklinde bir yansımadan bahsedilemez.

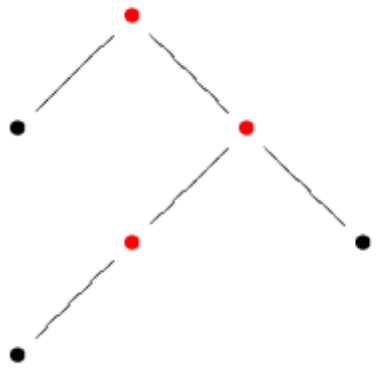
SORU 20: Internal Path Reduction Trees (İç Yol İndirgeme Ağaçları)

Bilgisayar bilimlerinde kullanılan veri saklama ve veriye kolay ulaşma yöntemlerinden birisi de ağaçlardır. Çok farklı şekillerde ağaçların kodlanması ve modellenmesi mümkündür. Bu özel ağaçlardan birisi de iç yol indirgeme ağaçlarıdır (internal path reduction tree, IPR Tree).

Bir ipr ağacı kısaca bir ikili ağaçtır (binary tree). Ayrıca IPR ağaçlarının dengeli olması şarttır (balanced tree).

Aynı şekilde ağacı sürekli olarak dengeli tutan ve bir ikili ağaç olan (binary tree) AVL ağaçları ile kıyaslandığında performans olarak eşit veya daha iyi olması söz konusudur. Yani IPR ağaçları AVL ağaçlarından (avl tree) kötü performansta olamazlar, iyi veya en kötü durumda eşit performans gösterirler.

IPR ağaçlarının derinlik hesaplamasında ağacın iç yol uzunluğu (internal path length) hesaplanır. Bu değerin hesaplanışını hatırlayacak olursak, ağaçtaki iç düğümlere erişmek için yapılan işlem sayısının toplamıdır. aşağıdaki temsili ağaçta iç yol uzunluğu 3′tür.



Tam bu noktada kaynaklardaki bir ayrılıktan bahsetmekte yarar vardır. Bazı kaynaklar kök düğüme (root node) erişimin maliyetini 0 kabul ederken bazı kaynaklar bu maliyeti 1 kabul etmektedir. Örneğin yukarıdaki örnekte şayet kök düğüme erişim maliyeti 0 kabul edilirse ağacın toplam iç yol uzunluğu

0+ 1 + 2 + 0 = 3

Olarak hesaplanır. Ancak bazı kaynaklar iç yol uzunluğunu hesaplarken kök düğümün maliyetini 1 olarak alıp bütün düğümlere (sadece iç düğümlere (internal nodes) değil, hem iç hem de dış düğümlere (external nodes) olan uzaklık) olarak hesaplamaktadır.

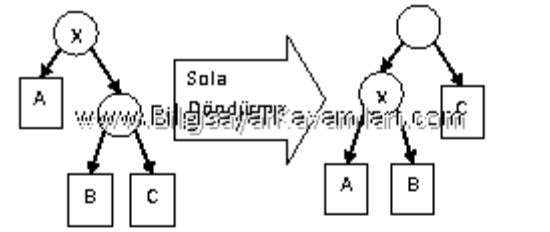
Bu durumda yukarıdaki ağacın iç yol uzunluğu:

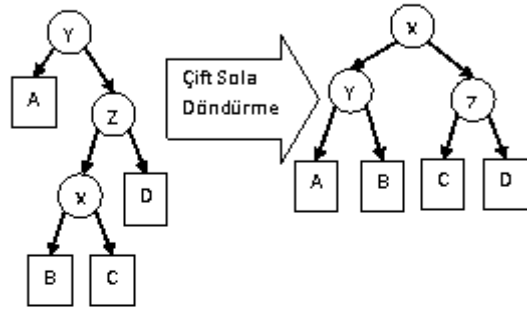
1 + 2 + 2 + 3 + 3 + 4 = 15

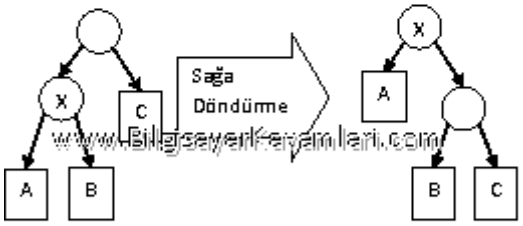
Olarak bulunur.

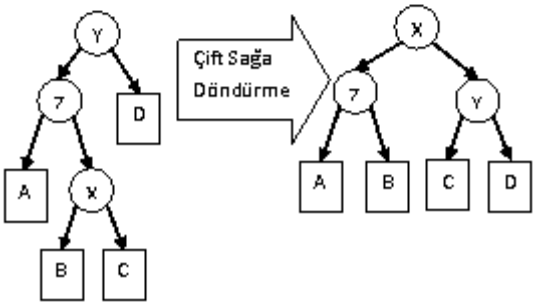
IPR ağacının çalışması sırasında istenen bir hesaplama yöntemi kullanılabilir, sonuç değişmemektedir.

IPR ağacı üzerinde yapılabilecek işlemleri aşağıdaki şekilde sıralayabiliriz:









Yukarıda sıralanan dört ayrı işlem, IPR ağaçlarında karşılaşılabilecek durumlardır. Bu durumları sıralayacak olursak, yukarıdaki 4 şekil için sırasıyla:

$n_c > n_b$ => sola döndür

$n_x > n_a$ => çift sola döndür

$n_c > n_a$ => sağa döndür

$n_x > n_a$ => çift sağa döndür

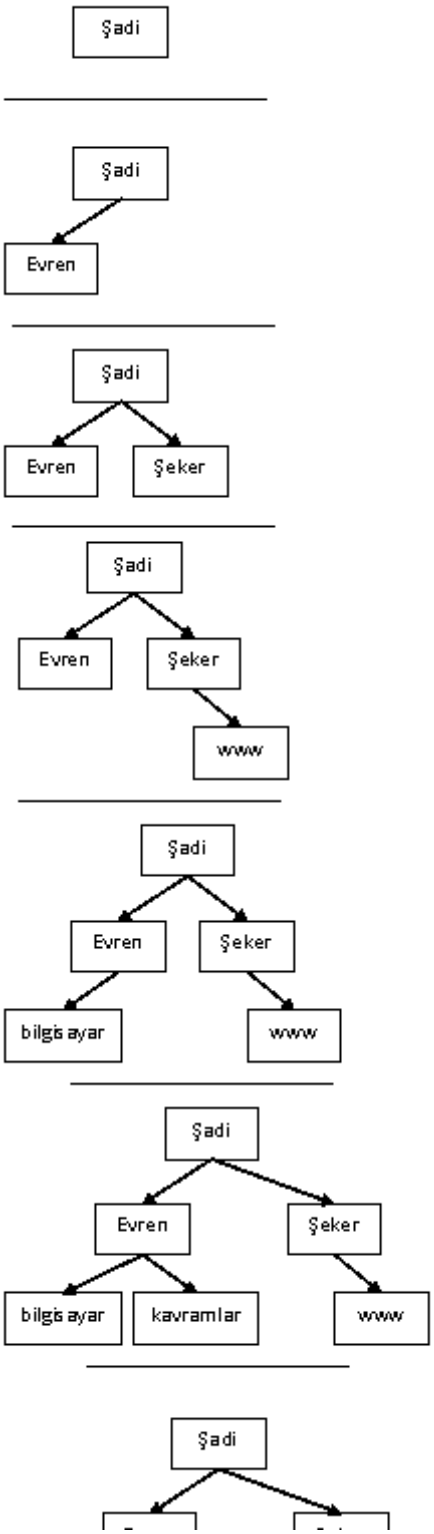
olarak sıralanabilir.

Yukarıdaki açıklamalardan sonra aşağıdaki kelimeleri bir ağaca yerleştirerek IPR ağacının nasıl çalıştığını görelim. Burada dikkat edilecek bir husus, IPR ağacının sadece yerleştirme (insertion) kuralının ikili arama ağacından (binary search tree) farklı olduğudur. Bunun dışında, ağaçta arama yapma işlemi veya bir bilgiyi değiştirme işlemi ikili arama ağacı (binary search tree) ile aynıdır.

Örnek olarak yerleştireceğimiz (insert) kelimeler aşağıdaki kelimeler olsun:

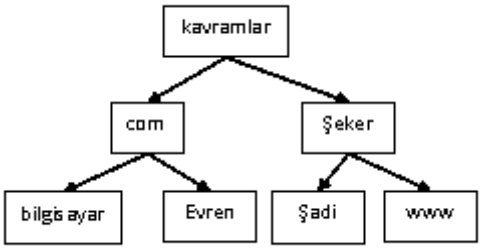
Şadi, Evren, ŞEKER, www, bilgisayar, kavramları, com

Yukarıdaki 7 kelimeyi verilen sırayla yerleştirmeye çalışalım:



Yukarıdaki ekleme işlemleri sırasında, klasik bir ikili arama ağacına ekleme dışında bir işlem yapılmamıştır. Ancak ağacın son halinde, eklenen son kelime (“com” kelimesi) ile birlikte ağacın dengesi bozulmuştur. Bu aşamaya kadar ağaçta bir dengesizlik bulunmamaktadır.

Ağacın son halindeki durum çift sağa döndürme gerektirip, döndürülmüş hali aşağıda gösterilmiştir:

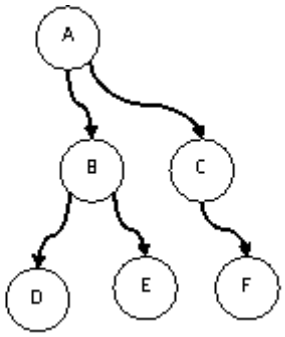


Bu döndürme işlemi iki aşamada yapılmıştır. Birinci aşamada çift sağa döndürürken x = kavramları, z = evren ve y = şadi şeklinde düşünülebilir. Ardından oluşan yeni ağacın hem solunda hem de sağında yeniden döndürme işlemleri icra edilmiştir.

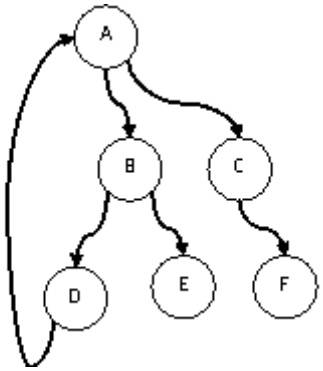
SORU 21: Sınırlı Derin Öncelikli Arama (Depth-Limited Search)

Bilgisayar bilimlerinde kullanılan arama algoritmalarından birisidir. Bu algoritma esas olarak derin öncelikli arama (depth first search DFS) ile aynı çalışmaktadır ancak tek farkı arama işlemi sırasında özellikle dairelere (cycles) takılma ihtimaline karşı sınır önlemi alınmış olmasıdır.

Örneğin aşağıdaki şekli ele alalım:



Yukarıdaki şekil tanım itibariyle bir ağaç özelliği göstermektedir. Yani yönlü ve daire içermeyen bir şekildir (directed acyclic graph). Ancak aynı şekle aşağıdaki gibi basit bir bağlantı daha eklenseydi artık ağaç olmayacaktı:



Yukarıdaki yeni şekilde derin öncelikli bir arama yaptığımızı ve A düğümünden işleme başladığımızı düşünelim. Örneğin orta sıra (infix) ve L N R (sol üst sağ , left node right)

sırasıyla arama yaptığımızı düşünelim. Daire içermeyen ilk şekilde LNR sırasına göre aşağıdaki sonucun çıkması beklenir:

DBEAFC

Ancak ikinci şekilde LNR sırasına göre önce en soldaki terim yazılmaya çalışılacak, Böylece A->B->D->A->B->D->A->B->D->A sırasıyla namütenahi dönülecek ve hiçbir zaman bitmeyecek bir fasit daireye girilecektir (Sonsuz döngü). Bu durum literatürde sol özyineleme (left recursion) olarak geçer. Yani şeklimizin (veya herhangi bir yapının) sol tarafında kendini tekrarlayan bir durum bulunmakta dolayısıyla derin öncelikli arama yapılamamaktadır.

Çözüm olarak bu yazının da konusu olan sınırlı derin öncelikli arama (depth limited search, DLS) algoritması kullanılabilir. Bu algoritmada gidilebilecek düğüm sayısına bir tahdit konulmakta ve ancak verilen sayıda düğüme gidilebilmektedir.

Algoritmanın kodlanması

Yukarıda izah edilen algoritma aşağıdaki şekilde kodlanabilir:

```
1   DLS(bakılan, hedef, azamiDerinlik)
2   {
3   //www.bilgisayarkavramlari.com
4     if (bakılan == hedef )
5       return bakılan;
6     yığın_koy(bakılan);
7     while (yığın doluyken )
8     {
9       geçici = yığın_al();
10      if(geçici.derinlik < azamiDerinlik)
11      {
12          DLS(geçici,hedef,azamiDerinlik);
13      }
14    }
15  }
```

Yukarıdaki özyineli fonksiyonda (recursive function) bakılan düğüm hedef olana kadar dolaşma işlemi devam etmektedir. Dolaşma işlemi sırasında klasik derin öncelikli aramalarda kullanılan yığın (stack) kullanılmış ve geçilen düğümler geri dönülüp aranmak üzere yığında tutulmuştur.

Şayet aranan düğüm verilen derinlikten daha derin değilse arama işlemi devam etmektedir ancak verilen derinlik geçildiği zaman arama işlemi daha derine gitmemekte ve artık o ana kadar aranmak üzere yığınladığı düğümleri işlemektedir.

Yukarıda anlatılan algoritma bilgisiz bir arama algoritmasıdır (uninformed search algorithm) ve ayrıca algoritmanın hafıza karmaşıklığı (memory complexity) sınırlıdır çünkü algoritmada aranabilecek düğüm sayısında bir sınır bulunmaktadır.

SORU 22: Arama Algoritmaları (Search Algorithms)

Bilgisayar bilimlerinde, çeşitli veri yapılarının (data structures) üzerinde bir bilginin aranması sırasına kullanılan algoritmaların genel ismidir. Örneğin bir dosyada bir kelimenin aranması, bir ağaç yapısında (tree) bir düğümün (node) aranması veya bir dizi (array) üzerinde bir verinin aranması gibi durumlar bu algoritmaların çalışma alanlarına girer.

Yapısal olarak arama algoritmalarını iki grupta toplamak mümkündür.

- Uninformed Search (Bilmeden arama)
- Informed Search (Bilerek arama)

Arama işleminin bilmeyerek yapılması demek, arama algoritmasının, probleme özgü kolaylıkları barındırmaması demektir. Yani her durumda aynı şekilde çalışan algoritmalara uninformed search (bilmeden arama) ismi verilir. Bu aramaların bazıları şunlardır:

- Listeler (diziler (array)) üzerinde çalışan arama algoritmaları:
  - Doğrusal Arama (Linear Search)
  - İkili arama (binary search)
  - Interpolasyon Araması (Ara değer araması, Interpolation Search)
- Şekiller (graflar (Graphs)) üzerinde çalışan algoritmalar
  - Sabit Maliyetli Arama (Uniform Cost Search)
  - Floyd Warshall algoritması
  - Prim's Algoritması
  - Kruskal Algoritması
  - Dijkstra Algoritması
  - Bellman Ford Algoritması
  - İkili arama ağacı (Binary Search Tree)
  - Prüfer dizilimi
  - Ağaçlarda Sığ öncelikli arama (breadth first search)
  - Şekillerde (Graph) sığ öncelikli arama (Breadth First Search, BFS)
  - Derin öncelikli arama (depth first search)
  - Derin Limitli Arama (Depth Limited Search) Algoritması
  - Yinelemeli Derinleşen Derin Öncelikli Arama Algoritması (Iterative Deepining Depth First Search, IDDFS)
  - Patricia Ağaçları
  - Trie Ağaçları (metin ağaçları, trie trees)
  - B-ağaçları (B-Tree)
- Metin arama algoritmaları (bir yazı içerisinde belirli bir dizgiyi (string) arayan algoritmalar)
  - Horspool Arama Algoritması
  - Knuth-Morris Prat arama algoritması
  - Boyer-Moore Arama algoritması
  - Kaba Kuvvet Metin Arama Algoritması (Brute Force Text Search, Linear Text Search)
  - DFA Metin Arama Algoritması

Arama işleminin bilerek yapılması ise, algoritmanın probleme ait bazı özellikleri bünyesinde barındırması ve dolayısıyla arama algoritmasının problem bazlı değişiklik göstermesi demektir. Bu algoritmaların bazıları a aşağıda listelenmiştir:

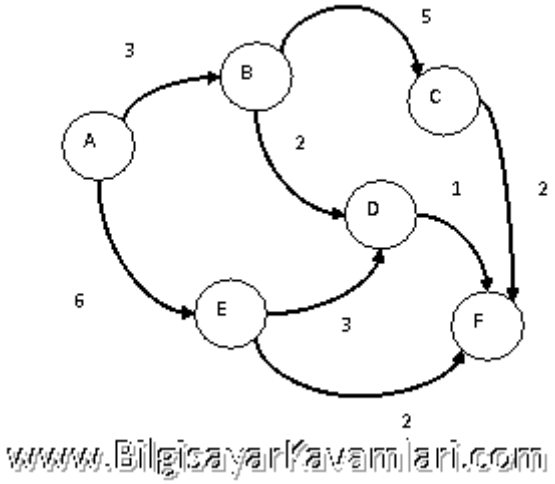
- Minimax Ağaçları
- Simulated Annealing (Benzetimli Tavlama) algoritması
- Tepe Tırmanma Algoritması (Hill Climbing Algorithm)
- Arı sürüsü arama algoritması (bees search algorithm)
- A* Araması (astar search)
- Geri izleme (backtracking)
- Işın arama (beam search)

SORU 23: Sabit Maliyet Araması (Uniform Cost Search)

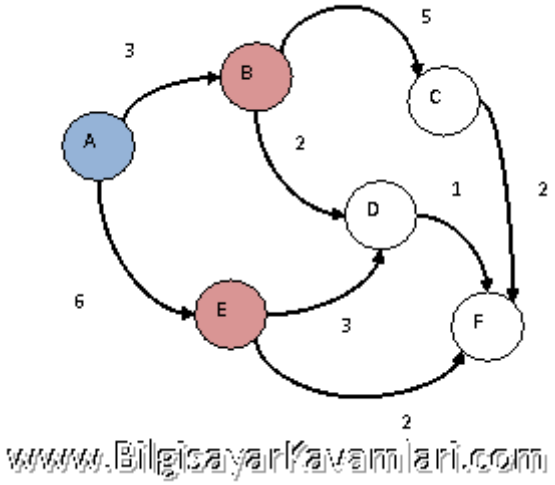
Bilgisayar bilimlerinde arama algoritmaları için kullanılan bir terimdir. Algoritma ağırlıklı graflar (weighted graphs) üzerinde çalışmaktadır. Ağaçlar da bir graf örneği olduğu için algoritmanın ağaçlar üzerinde çalışması da mümkündür. Algoritma basitçe aşağıdaki şekilde tanımlanabilir:

1. Kök düğümden başla (root node)
2. En düşük maliyetli komşuya git
3. Şayet aranan düğüm bulunduysa bit, bulunmadıysa 2. Adıma geri dön

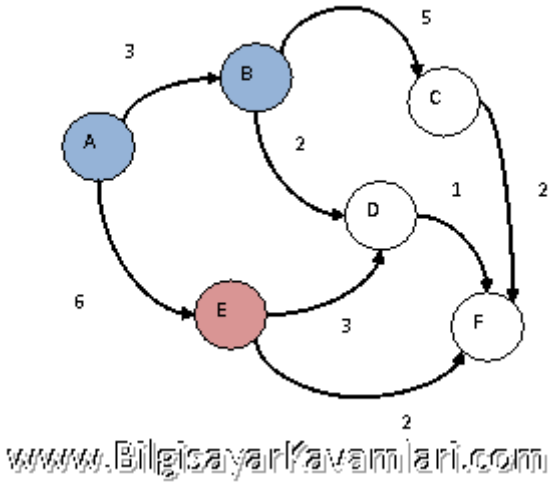
Yukarıdaki 3 basit adımla açıklanabilen algoritmanın çalışmasını aşağıdaki temsili graf üzerinden açıklamaya çalışalım:



Yukarıdaki şekilde A düğümünden başladığımızı ve F düğümüne ulaşmak istediğimizi düşünelim. Algoritma ilk başta A düğümünün komşularının maliyetini çıkarır ve en az maliyetli komşuya gider.

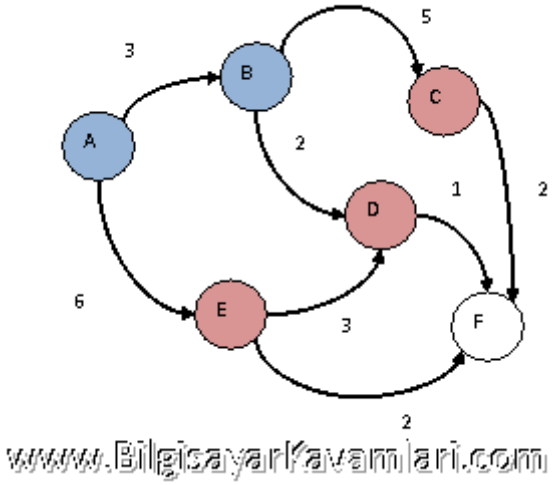


Yukarıda gösterildiği üzere A düğümünün komşusu iki düğüm vardır. B ve E düğümleri bu sayede kontrol edilmiş olunur.

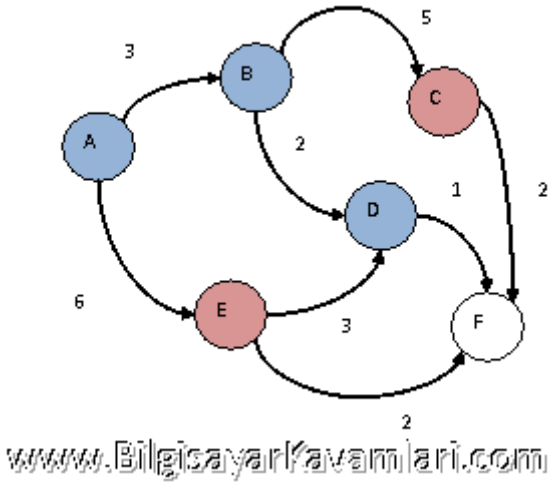


Yukarıdaki şekilde mavi gösterilen düğümler gittiğimiz, kırmızı olanlar ise baktığımız düğümlerdir. Yani ilk adımda B ve E düğümlerine bakılır bu düğümlerden kısa olan B düğümüne gidilir.

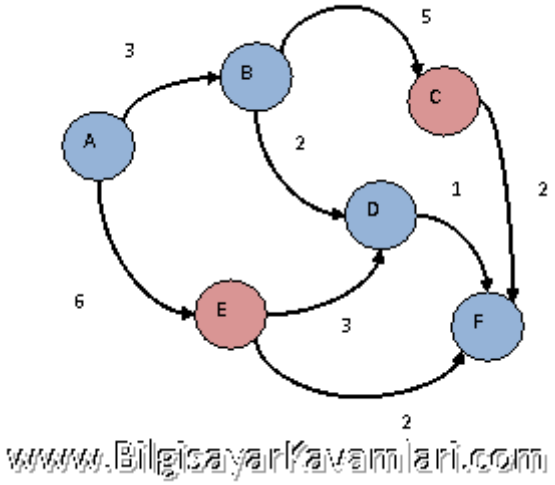
B düğümünden yine komşu düğümler kontrol edilir.



Böylelikle C ve D düğümlerini de kontrol etmiş oluruz. Bu düğümlerden kısa mesafeli olan D düğümü seçilir.



Son olarak D düğümünün komşularına bakıldığında hedef olan F düğümü bulunur ve bu düğüm seçilerek algoritma bitirilir.



Sonuçta arama algoritmamız, A,B,D ve F düğümlerini dolaşarak hedefe varmış olur.

Yukarıdaki örnek, yönlü ve ağırlıklı graf üzerinden anlatılmıştır (weighted directed graph) aynı algoritma yönsüz ve ağırlıklı bir grafik üzerinde de çalışır.

Ayrıca yukarıdaki örnekte az sayıda düğüm olduğu için bütün düğümlere bir kez bakılmıştır. Ancak sabit maliyetli arama algoritmasında bu her zaman gerçekleşmek zorunda değildir. Yani arama algoritmamızın sonucu bulduğu ama grafta bakmadığı düğümler olduğu durumlar olabilir.

Sabit maliyetli arama algoritmasını aslında bir öncelik sırasına (priority queue) benzetmek mümkündür. Algoritma başlangıç düğümünden başlayarak graftaki bütün düğümleri, başlangıç düğümüne olan mesafelerine göre dolaşır.

Örneğin yukarıdaki graf için her düğümün başlangıca uzaklığı yazılacak olursa:

A	0
B	3

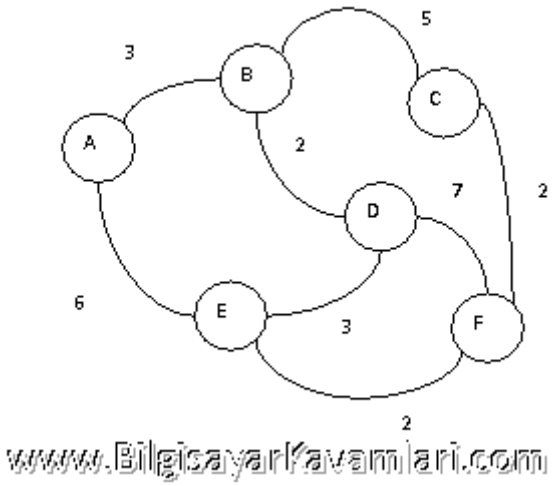
C	8
D	5
E	6
F	6

Uzunluklarına sahiptir ve bu durumda düğümler uzaklıklarına göre sıralandığında

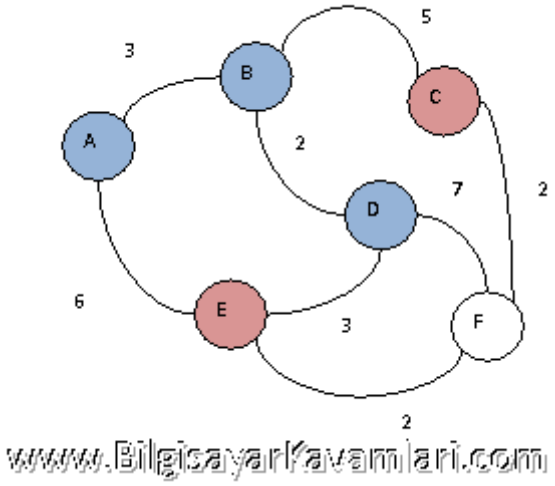
A,B,D,(E,F),C

Sırası çıkar ki bu sıralamada F düğümüne ulaşan yol yukarıdaki arama sonucunda bulunan A,B,D,F sırası olur.

Sadece bu algoritmayı daha iyi anlayabilmek için yukarıdaki graf üzerinde ufak bir değişiklik yapalım ve D-F aralığının maliyetini 7′ye çıkaralım. Ayrıca grafın yönsüz olduğunu kabul edelim.



Yukarıdaki yeni haliyle grafımız bir öncekine göre daha ilginç çalışacaktır. Algoritmamız gereği yine A düğümünden başlayacağız ve D düğümüne gelene kadar aynı şekilde arama işlemi devam edecektir. Çünkü D düğümüne ulaşana kadar ne algoritmada ne de grafta bir farklılık yoktur.



Yukarıdaki şekilde D düğümüne ulaşıldıktan sonra F düğümüne giden yolun maliyeti yüksek bulunacaktır. Bunun sebebi D düğümünden F düğümüne ulaşan yolun maliyetinin toplamda A-B + B-D + D-F maliyetlerinden oluşması ve bunun da 3 + 2 + 7 = 12 olmasıdır.

Algoritmamız bu değere ulaştıktan sonra A,B,D,E ve F yolunu deneyecek, bu maliyeti bir öncekine göre daha ucuz bulacaktır. 3+2+3+2 = 10

Ancak E düğümüne ulaşma maliyeti A-E dolaşması sırasında zaten 6 olarak bulunmuştu. Bu durumda algoritmamız yeni yol olarak A,E,F şeklinde sonucu bulacaktır ve maliyeti 6+2 = 8 olarak en kısa yola sahiptir.

Yukarıdaki bu yeni yaklaşımda geri izleme algoritması (back tracking algorithm) kullanılmıştır.

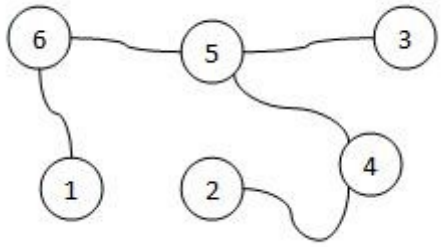
SORU 24: Prüfer Dizilimi (Prüfer Sequence)

Bilgisayar bilimleri de dahil olmak üzere pek çok bilim ve mühendislik alanının ortak çalışma konularından birisi olan graf teorisindeki (graph theory) bir hesaplama veya gösterim algoritmasıdır.

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

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

Örneğin aşağıdaki şekli (graph) ele alalım:



şekildeki grafta toplam 11 düğüm (node) bulunmaktadır ve şekilde graf yönsüz graftır (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 grafı yapraklardan köke doğru artan sayılar ile bu yüzden numaralandırdık.

Sırasıyla her düğümün bağlı olduğu diğer üst düğümü yazıyoruz.

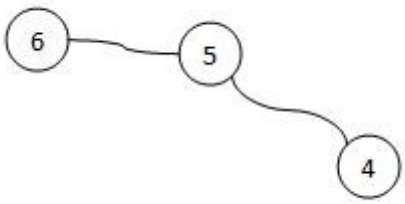
Bu durumda aşağıdaki düğümler yapraklardır ve bağlı oldukları üst düğümler yanlarında verilmiştir:

1 → 6

2 → 4

3 → 5

Yukarıdaki yapraklardan başlayarak bu dizilimi not ediyoruz. Yani şimdilik prüfer dizilimimizde 645 bulunuyor. İkinci adımda bu yaprakları şekilden (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 şekli (graph) geri çizebiliriz.

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

SORU 25: Kirchoff Teoremi (Kirchoff Theorem)

Bilgisayar bilimlerinin de arasında bulunduğu pek çok bilim ve mühendislik alanında kullanılan graf teorisinde kullanılan bir teoremdir. Bu teorem kirchoff matrisi veya laplas matrisi (laplacian matrix) ismi verilen matrisler ile birlikte kullanıldığında bir grafta bulunan asgari tarama ağacı (minimum spanning tree) sayısını verir.

Bilindiği (veya ilgili yazıdan okunabileceği) üzere laplas matrisleri diyagonda graftaki düğümlerin (nodes), düğüm derecelerini (node order) ve geri kalan elemanlarda da düğümlerin komşuluk listelerini (adjacency list) veren matrislerdir.

Kirchoff teoremine göre bir graftaki n tane düğüm (node) için laplas matrisinden çıkarılan ve sıfırdan farklı olan λ_1 , λ_2 , … , λ_{n-1} şeklinde özdeğerler (eigen values) bulunsun. Bu durumda bu grafta bulunabilecek asgari tarama ağacı (minimum spanning tree) sayısı aşağıdaki şekilde bulunur:

farklı asgari tarama ağacı sayısı = $(\lambda_0 \lambda_1 \lambda_2 \dots \lambda_{n-1})/n$

Yukarıdaki bu değer aynı zamanda laplas matrisinin herhangi bir kofaktör (cofactor) değerinin mutlak değerine eşittir.

SORU 26: Laplas Matrisi (Laplacian Matrix)

Bilgisayar bilimlerinin de içinde bulunduğu pekçok bilim ve mühendislik alanında kullanılan graf teorisi (graph theory) açısından önemli bir matristir. Laplas matrisinin özelliği her düğümün derecesini (node order) ve diğer düğümlerle olan komşuluk ilişkisini (adjacency list) tutmasıdır.

Laplas matrisine giriş matrisi (admittance matrix) veya Kirchhoff matirisi (Kirchhoff matrix) isimleri de verilmektedir. Kirchhoff teorsi (Kirchhoff theory) ile birlikte kullanılınca bir graftaki birbirinden farklı asgari tarama ağacı (minimum spanning tree) sayısını elde etmekte kullanılabilir.

Laplas matrisinin tanımı

Laplas matrisi bir grafta bulunan düğüm sayısı kadar boyutlara sahip kare matristir. Düğüm sayısı n olan bir graf için nxn boyutlarında bir matris aşağıdaki şartlara göre üretilir.

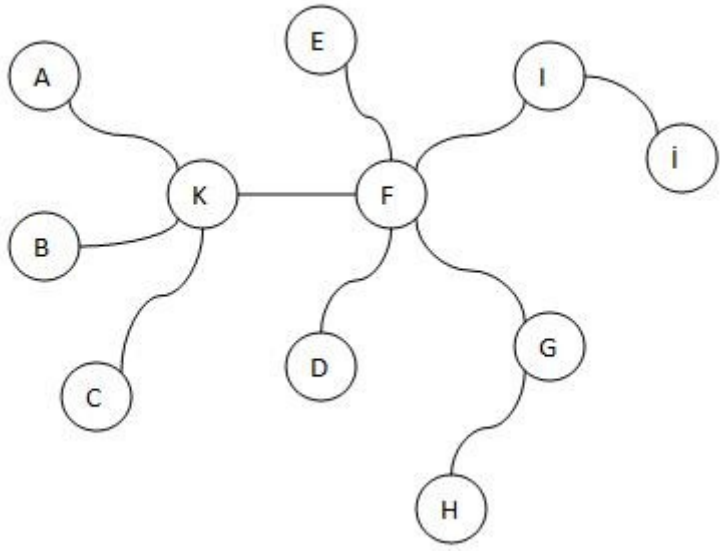
i=j -> düğümün derecesi (node order)

i≠j -> i. düğümün j. düğümle olan komşuluğu (komşuluk varsa -1 yoksa 0)

Yukarıdaki kurallardan anlaşılacağı üzere matrisin köşegeninde (diagon) düğüm dereceleri ve matrisin geri kalanında ise -1 ve 0'dan oluşan komşuluk matrisi bulunacaktır.

Laplas matrisine örnek

Örneğin aşağıdaki grafın laplas matrisini oluşturmak isteyelim:



Yukarıdaki şekilde 11 düğüm bulunmaktadır. Bu durumda 11×11 boyutlarında bir matris (masfuf) oluşturarak tanımımızda verilen kurallara uygun bir şekilde matrisin değerlerini dolduralım:

	A	B	C	D	E	F	G	H	I	J	K
A	1	0	0	0	0	0	0	0	0	0	-1
B	0	1	0	0	0	0	0	0	0	0	-1
C	0	0	1	0	0	0	0	0	0	0	-1
D	0	0	0	1	0	-1	0	0	0	0	0
E	0	0	0	0	1	-1	0	0	0	0	0
F	0	0	0	-1	-1	5	0	0	-1	0	-1
G	0	0	0	0	0	-1	2	-1	0	0	0
H	0	0	0	0	0	0	-1	1	0	0	0
I	0	0	0	0	0	-1	0	0	2	-1	0
J	0	0	0	0	0	0	0	0	-1	1	0
K	-1	-1	-1	0	0	-1	0	0	0	0	4

Yukarıdaki tablodan açıkça görülebileceği üzere laplas matrisinin bir özelliğide satır bazında ve sütün bazında toplamların 0 olmasıdır.

Laplas matrisinin özellikleri

G ismindeki bir graf ve L ismindeki bir Laplas matrisinin $\lambda_0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_{n-1}$ koşullarını sağlayan özdeğerleri (eigen values) için aşağıaki özellikler sayılabilir:

Şayet laplas matrisinin bütün özdeğerleri (eigenvalues) sıfırdan büyükse bu matris için Hermitian matris denilebilir

λ_0 her zaman 0′dır.

λ_1 matematiksel bağlılığı (algebraic connectivity) gösterir

Özdeğerlerin 0 olma sayısı G grafındaki bağlı düğüm sayısını verir.

SORU 27: Düğüm Derecesi (Order of Node)

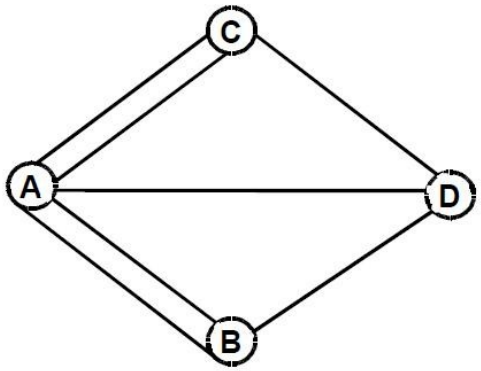
Graf teorisinde (graph theory) bir grafın temel unsurlarından olan düğümlerin (nodes) giren veya çıkan kenar (edge) sayısını verir.

Tanım olarak yönsüz graflar (undirected graphs) ve yönlü graflar (directed graphs) için iki ayrı tanım yapılabilir.

Yönsüz graflarda bir düğümün derecesi

Yönsüz graflarda bir düğümün derecesi, doğrudan düğüme bağlı komşu düğüm sayısına veya o düğüme bağlı kenar sayısına eşittir.

Örneğin aşağıdaki yönsüz grafı (undirected graph) ele alalım:



Yukarıdaki bu grafta bulunan düğümlerin dereceleri aşağıda listelenmiştir:

A 5

B 3

C 3

D 3

Görüldüğü üzere düğümlerin dereceleri komşu olduğu düğümlerin sayısı veya o düğüme bağlı olan kenar sayısı olarak bulunabilir.

Yönlü graflarda düğüm derecesi

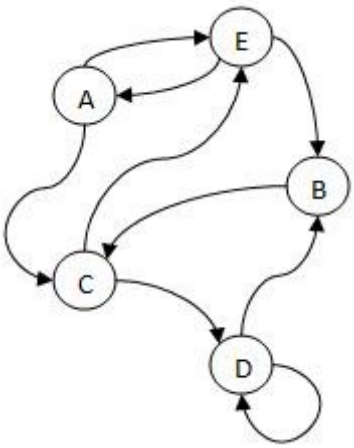
Yönlü graflarda(directed graphs) düğüm derecesinden bahsedilirken iki farklı değer bulunabilir:

- Giren derecesi (Inorder)
- Çıkan derecesi (outorder)

Bir düğümün giren derecesi yönü düğüme doğru olan kenar sayısıdır. Aynı zamanad bir düğüme ulaşılabilen diğer düğümlerin sayısıdır.

Çıkan derecesi ise bir düğümden çıkan kenarların sayısı veya bir düğümden ulaşılabilen diğer düğümlerin sayısı olarak tanımlanır.

Daha iyi anlaşılması için aşağıdaki grafı ele alalım:



Yukarıda gösterilen yönlü graftaki düğümlerin giren dereceleri (in order) aşağıdaki şekilde sıralanabilir:

A 1

B 2

C 2

D 2

E 2

Benzer şekilde çıkan dereceleri (out order) de aşağıda verilmiştir:

A 2

B 1

C 2

D 2

E 2

Dikkat edilirse bir graftaki çıkan dereceleri ile giren derecelerinin toplamı eşit olmalıdır. Örneğimizde 9 = 9 durumu doğrudur.

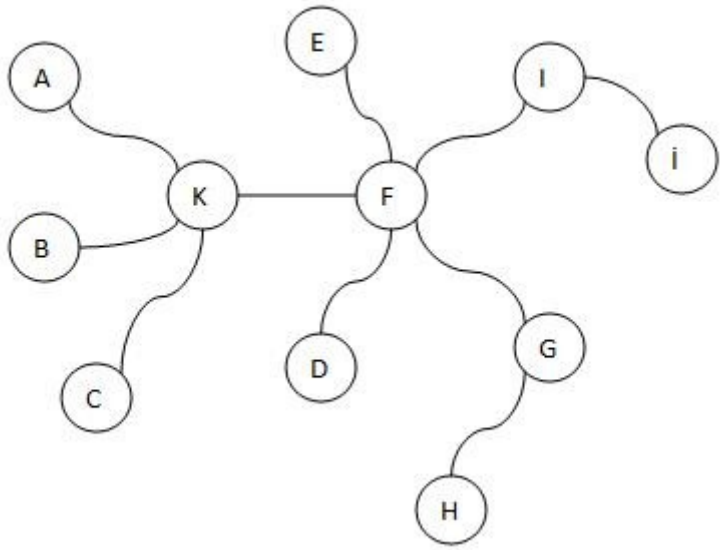
SORU 28: Denkşekillilik (Isomorphism)

İki şeklin birbirinden farklı ancak denk olması durumudur.

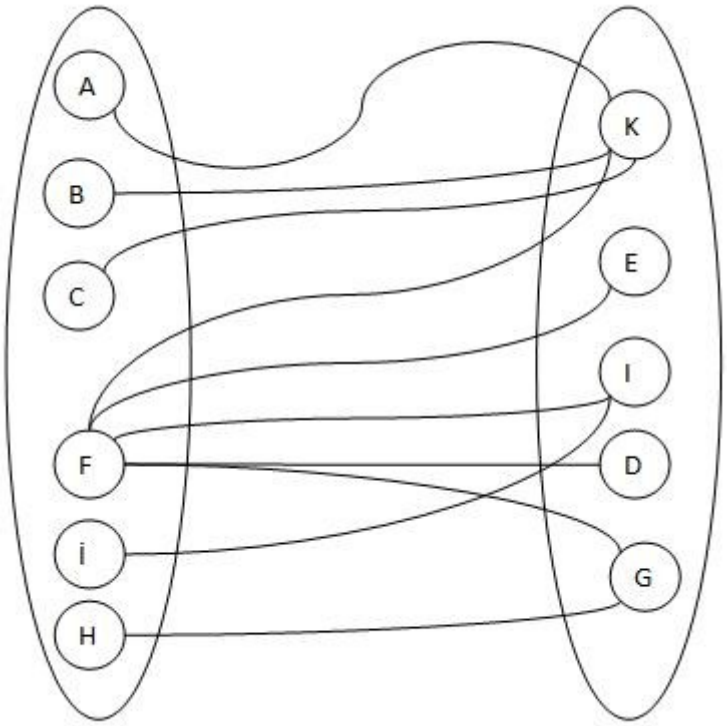
Bilgisayar bilimleri de dahil olmak üzere pek çok bilim ve mühendislik alanında kullanılan graf teorisine (graph theory) göre iki şekil birbirinden farklı çizilmiş ancak işlev ve değer olarak aynı olabilir.

Tanım ve örnek

Örneğin aşağıdaki iki şekli ele alalım:



Yukarıda verilen graftaki düğümler (nodes) ik ifarklı 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 graf denkeşli ise bu grafların birisndeki döngü (cycle) sayısı diğerine eşittir.

Şayet iki graf denkeşli ise bu graflardan 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 $\supset$ 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.

SORU 29: Öyler Yolu (Eulerian Path)

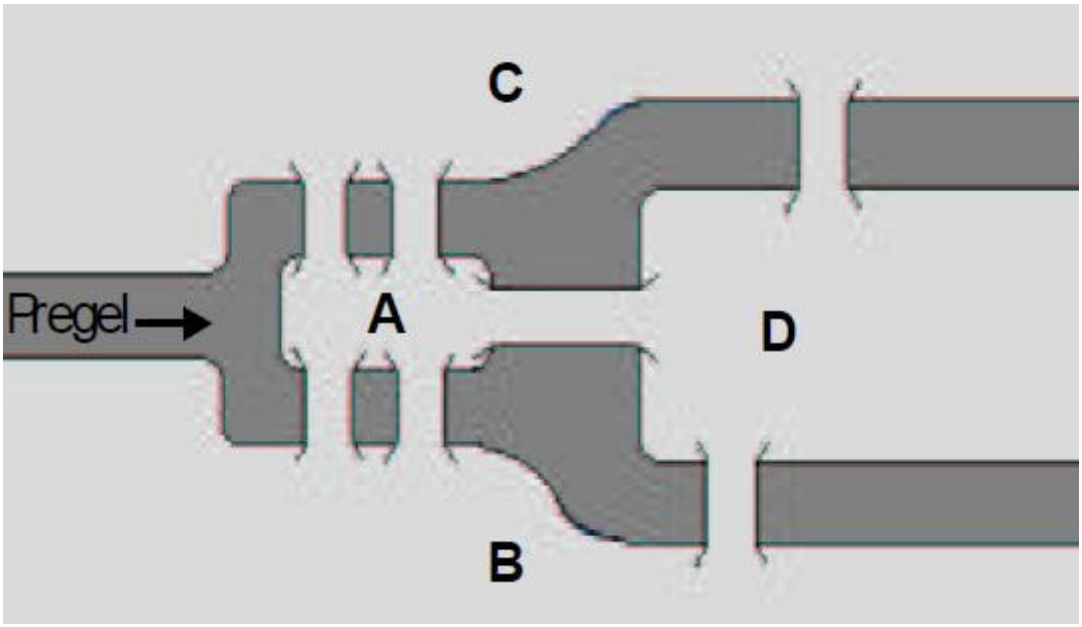
Bilgisayar mühendisliği de dahil olmak üzere pekçok bilim ve mühendislik alanında kullanılan graf teorisindeki özel bir yol (path) şeklidir. Bu yolun özelliği her kenardan (edge) bir kere (en az ve en çok) geçen yolu bulmaktır.

İçerik

İlk bakışta hamilton yolu (hamiltonian path) ile karıştırılabilir ancak Hamilton yolundaki amaç her düğümden (node) bir kere geçen yolu bulmak iken Öyler yolunda her kenardan (edge) bir kere geçen yolu bulmak amaçlanır.

Öyler teorisinin tarihi çıkışı

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.

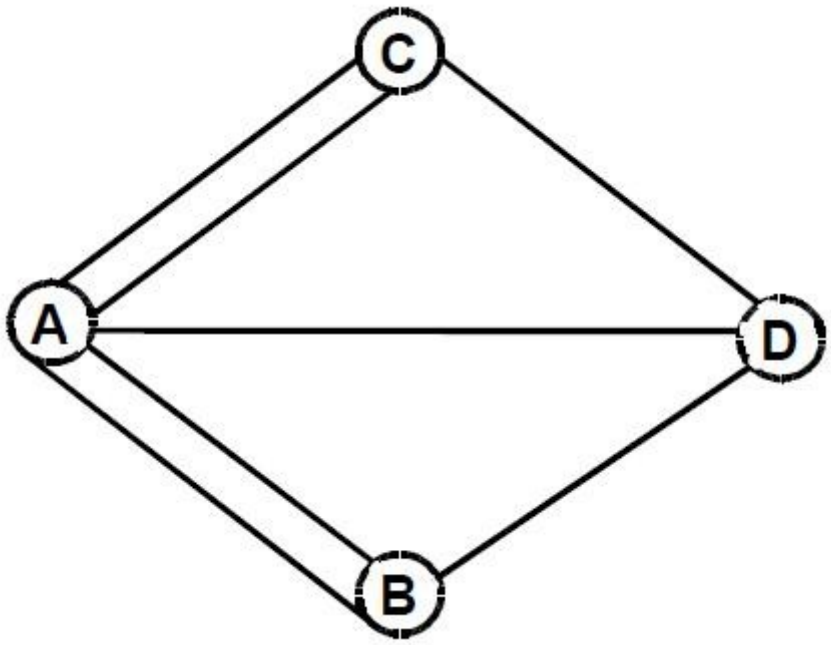
Problem bütün köprülerden bir kere geçilen bir yol olup olmayacağıdır.

Öyler bu soruyla uğraşırken yazımızın konusu olan öyler yolu teorisini bulmuştur ve cevap olarak böyle bir yolun bulunamayacağını istaplamıştır.

Öylerin iddiası bastir 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 graf 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 grafta (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 grafa ise yarı öyler (semi-eulerian) veya dolaşılabilir (traversable) graf 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. Bu yolu içeren grafa ise öyler grafı (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.

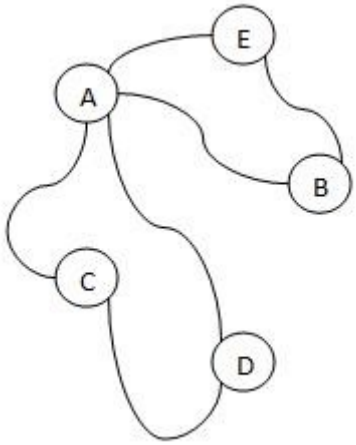
Öyler yolunun özellikleri

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

Öyler Döngülerinin sayısı

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

Örneğin aşağıdaki 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.

Buradaki soru acaba bir grafta 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 grafta bulunan öyler döngülerinin sayısı graftaki bütün düğümlerin derecelerinin bire eksiğinin faktöriyellerinin çarpımına eşittir.

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

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

Örneğin yukarıdaki graf 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.

SORU 30: Markof Modeli (Markov Model)

Bilgisayar bilimleri de dahil olmak üzere pekçok bilim ve mühendislik alanında kullanılan markof modelleri aslında graf teorisinin (graph theory) bir uygulamasıdır.

Basitçe düğümleri (nodes) durumlardan oluşan ve bu durumlar arasında istatistiksel geçişi modelleyen kenarları (edges) bulunan graflardır.

Markof modellerine (markof zinciri (markov chain) ismi de kullanılmaktadır) göre bir durum belirli bir istatistiksel değere göre değişir veya değişmeden aynı kalır. Ayrıca geçmiş durumların mevcut durum üzerinde bir etkisi söz konusu değildir. Ancak şimdiki durum gelecek durumları etkileyebilir.

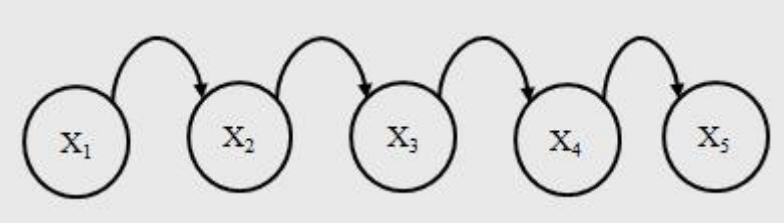
Markof modelinin tanımı

Markof modellerinin istatistiksel olma özelliğinden dolayı her bir stokastik olayın (Stochastic Event) olasılık değerini modelleyen bir gösterimi mümkündür.

Bu olasılıkların gösterildiği formül

$$P[X_{t+1} = x_{t+1} \mid X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_1 = x_1, X_0 = x_0]$$
$$= P[X_{t+1} = x_{t+1} \mid X_t = x_t] \text{ olarak tanımlanır.}$$

Yukarıdaki formülde t+1 zamanındaki olayların t zamanına bağlı olması söz konusudur. Yukarıdaki bu zaman bağlamında bağımlılıktan dolayı markov modellerinin yönlü graf olmaları gerekir.



Yukarıdaki şekilde bu olaylar arası bağlantı yönlü graf (directed graph) olarak görülmektedir. Elbette yukarıdaki graf sadece bir örnek olup olayların doğrusal olarak bağlanmaları gerekemez ancak yukarıdaki graftan anlaşılması gereken her olayın bir sonraki olaya belirli bir olasılık değeriyle devam edebileceği veya aynı kalacağıdır. Örneğin t anında X1 olayı oluyor olsun. t+1 zamanında X1 devam edebilir veya X2 olayına geçilebilir.

Yukarıdaki bu gösterimlere ilave olarak markov zincileri birer masfuf (matrix) ile de gösterilebilir.

	A	B	C	D
A	0.95	0	0.05	0
B	0.2	0.5	0	0.3
C	0	0.2	0	0.8
D	0	0	1	0

Örneğin yukarıdaki matriste 4 olay arasındaki geçiş değerleri gösterilmiştir. Matrisin tek yönlü olmasına dikkat edilebilir. Yani köşegen simetriği (diagonal symmetry) mutlaka 0 olmalıdır. Örneğin C'den B'ye olasılık değeri 0.2 iken B'den C'ye 0 olmaktadır. Ayrıca satır bazında ihtimallerin toplamları 1 olmalıdır. Yani satırın ifade ettiği stokastik olaydan farklı bir stokastik olaya geçişin veya aynı olayda kalmanın ihtimalleri toplamı 1 olmalıdır.

Markof zincirlerine hava durumu örneği

Markof zincirleri tahmin (forecasting) için oldukça kullanışlıdırlar. Örneğin hava durumu tahmini için markof zinciri kullanmak isteyelim ve iki olayımız olsun:

- bugünkü hava
- yarınki hava

şimdi elimizde bugünkü havanın durumu bulunmakta. Bu olaydan yola çıkarak yarınki hava olayını (tahminini) yapmaya çalışalım ve olayı markof modeli ile modelleyelim.

- Bugün yağmur yağıyorsa -> yarın yağmur yağma ihtimali = 0.4
- Bugün yağmur yağıyorsa -> yarın yağmur yağmama ihtimali = 0.6
- Bugün yağmur yağmıyorsa -> yarın yağmur yağma ihtimali = 0.2
- Bugün yağmur yağmıyorsa -> yarın yağmur yağmama ihtimali = 0.8

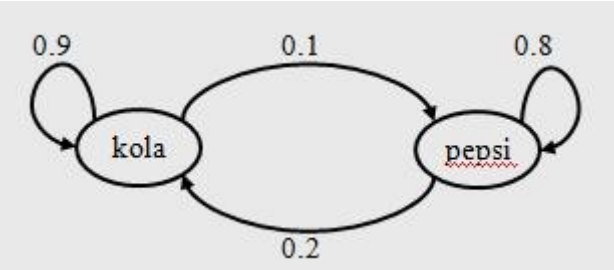
olarak verilmiş olsun. Bu bilgiyi geçmiş tecrübelerden edindiğimizi ve markof modeli ile modellemek istediğimizi düşünelim.

$$P = \begin{pmatrix} 0.4 & 0.6 \\ 0.2 & 0.8 \end{pmatrix}$$

Sonuç olarak yukarıdaki şekilde bir olasılık matrisi elde edilir. Bu matrise, stokastik matris (stokhastic matrix) ismi de verilmektedir.

Markof zincirleri ile Kola ve Pepsi örneği

Markof zincirlerinin anlatımı sırasında kullanılan meşhur örneklerden birisi de kola ve pepsi örneğidir. Bu örnekte bir kişinin en son aldığı içeceğin kola (coca cola) olması veya pepsi olması durumuna göre bir sonraki içeceğinin tahmin edilmesine çalışılır.



Örneğin bu iki içecek arasındaki ilişki ve karar olasılıkları yukarıdaki şekilde verilmiş olsun. Yani pepsi alan bir kişinin bir sonraki içceğinin yine pepsi olma olasılığı 0.8 ve kola olma olasılığı 0.2, benzer şekilde kola içen birisinin bir sonraki içeceğinin yine kola olma olasılığı 0.9 ve pepsi olma olasılığı 0.1 olarak verilsin.

Şimdi sorumuzu soralım:

Pepsi içen bir kişinin ikinci alış verişinde kola alma olsaılığı nedir?

Bu sorunun cevabı için stokastik matrise başvurarak basit bir matris çarpım işlemi yapabiliriz:

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}$$

Önce yukarıdaki şekilde stokastik matrisimizi çıkaralım ve olasılıkları yerleştirelim. Şimdi sorumuza dönecek olursak bize ikinci alış verişteki ihtimal sorulmuşt. Bu durumda matrisin karesini alalım (kendisi ile çarpalım)

$$P^2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix} \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix} = \begin{bmatrix} 0.83 & 0.17 \\ 0.34 & 0.66 \end{bmatrix}$$

Yukarıdaki p^2 matrisinden anlaşılacağı üzere markof modelimizdeki bir olayın tekrar etme olasılığını bulmuş olduk. Kişinin ikinci alışverişinde pepsiden kolaya geçme olasılığı yukarıdaki şekilde 0.34 olarak bulunmuş olur.

Örneğin yukarıdaki bu bilgiler ışığında bize şöyle bir soru da sorulabilir di:

Mevcut durumda insanların %60′nın kola ve %40′ının pepsi içtiklerini düşünelim. Bu insanların haftalık olarak içecek aldıklarını düşünürsek üç hafta sonra insanların ne kadarı kola içiyor olacaktır?

Şimdi bu soruyu çözerken 3 satın alma işlemi yani 3 kere markof modelde değişim işlemi yapılacağını hatırlayalım. Ardından mevcut durumun etkisini de ekleyerek hesabımızı aşağıdaki şekilde yapalım:

$$P(X_3 = 0) = \sum_{i=0}^{1} Q_i p_{i0}^{(3)} = Q_0 p_{00}^{(3)} + Q_1 p_{10}^{(3)} = 0.6 \cdot 0.781 + 0.4 \cdot 0.438 = 0.6438$$

Yukarıdaki hesapta öncelikle 3. satın alma işlemi sırasında stokastik matrisin değeri hesaplanmıştır. Yani P^3 için matris 3 kere kendisi ile çarpılmıştır. Ayrıca matristen elde edilen katsayılar 0.6 ve 0.4 mevcut durum oranıyla çarpılmıştır. Sonuçta 0.6438 oranında kişinin kola içeceği bulunmuş olunur.

SORU 31: Floyd-Warshall Algoritması

Bilgisayar bilimlerinin önemli konularından olan algoritma analizi sırasında sıkça bahsi geçen bir algoritmadır. Algoritmanın ana amacı belirli bir graf üzerinde bir başlangıçtan(source) bir bitiş düğümüne (sink, end, target) en kısa yoldan (shortest path) ulaşmaktır.

Bu özelliğinden dolayı, maksimum akış (maximum flow) problemleri olarak bilinen, ve örneğin bir dağıtım şebekesinde bir kaynaktan bir hedefe gönderilebilecek azami miktarı sevk eden problemlerin çözümünde kullanılabilen bu algoritma, algoritmayı bulan iki kişinin ismi ile anılmaktadır.

İçerik

Bu algoritmanın ana amcı dinamik programlama (dynamic programming) konusunu daha iyi anlayabilmektir.

Algoritma

Algoritma basitçe bir grafta gidilebilecek düğümlerin komşuluk listesini (adjecency list) çıkararak bu düğümlere olan mesafeyi tutan bir masfuf (matris) üzerinden çalışır.

Algoritmanın her adımında matris üzerinde çeşitli işlemler yapılarak en kısa yol bulunmaya çalışılır.

Algoritma matris üzerinde çalışan ve düğümler arası mesafeleri her adımda güncelleyen bir algoritma olduğu için itartif (iterative, döngü ile) yazılabilir. Aşağıda müsvette kodlar (pseude codes) verilmiştir.

```
 1 int yol[][];
 2 /* Sonucun içinde bulunacağı ve anlık olarak bir düğümden
 3    diğerine ne kadar maliyet ile bulunduğunu tutan matris.
 4    İlk olarak komşuluk listesini tutar ve doğrudan
 5 gidilemeyen düğümlere olan mesafe sonsuz olarak tutulur.*/
 6
 7 procedure FloydWarshall ()
 8    for k: = 1 to n
 9       for each (i,j) in{1,..,n}²
10          yol[i][j] = min ( yol[i][j], yol[i][k]+yol[k][j] );
```

Yukarıdaki algoritmada görüldüğü üzere n adet düğümlü bir graf için iki boyutlu bir dizi oluşturulur. Bu dizinin içerisindeki değerlere ilk olarak komşuluk listesindeki değerler atanır ve doğrudan ulaşılamayan düğümler için sonsuz değeri doldurulur.

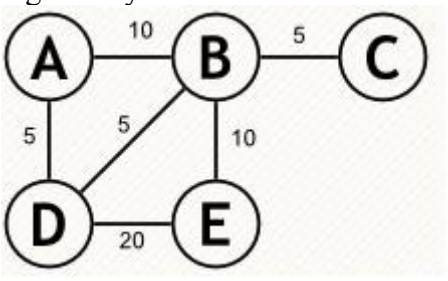
Ardından Matrisi dolaşan bir döngü ile (i ve j) yollar güncellenir. Bu matrisin taranması işlemi düğüm sayısı (n) kadar tekrar eder (yukarıdaki k döngü değişkeni tekrara yaramaktadır). Bu tekrar aslında üzerinden atlanma ihtimali olan düğümü belirtmektedir. Yani örneğin k = 3 için 3. düğüm marifetiyle ulaşılan düğümler belirlenir. (örneğin 1. düğümden 4. düğüme doğrudan erişim yoksa ama 1. düğümden 3. düğüme ve 3. düğümden de 4. düğüme erişim varsa)

Yukarıdaki algoritma daha basit bir şekilde yazılabilir:

```
for i = 1 to N
   for j = 1 to N
      if(i'den j'ye bir yol varsa)
         yol[0][i][j] = i ile j arasındaki mesafe
      else
         yol[0][i][j] = sonsuz

for k = 1 to N
   for i = 1 to N
      for j = 1 to N
         yol[k][i][j] = min(yol[k-1][i][j], yol[k-1][i][k]
+ yol[k-1][k][j])
```

Algoritmanın çalışmasına örnek

Algoritmanın çalışmasını daha iyi anlayabilmek için aşağıdaki örnek üzerinden adım adım algoritmayı kullanarak en kısa yolu bulalım:



Yukarıdaki şekil incelendiğinde A'dan E'ye giden birden çok yol bulunabilir:

```
Yol 1:    A -> B -> E            20
Yol 2:    A -> D -> E            25
Yol 3:    A -> B -> D -> E       35
Yol 4:    A -> D -> B -> E       20
```

Yukarıdaki yollar çıkarıldıktan sonra en kısasının 20 uzunluğunda olduğu bulunabilir. Şimdi bu yollardan en kısasını Floyd-Warshall algoritmasının nasıl bulduğunu adım adım inceleyelim:
1. Adımda komşuluk listesine göre matris inşa edilir.

Yukarıdaki şekilde doğrudan ilişkisi bulunan düğümler ve ağırlıkları aşağıda verilmiştir:

	A	B	C	D	E
A	0	10	∞	5	∞
B	10	0	5	5	10
C	∞	5	0	∞	∞
D	5	5	∞	0	20
E	∞	10	∞	20	0

Yukarıdaki grafta doğrudan ilişkisi bulunmayan düğümlerin değerleri ∞ olarak gösterilmektedir. Diğer değerler doğrudan ağırlıkları göstermektedir.

Şimdi algoritmanın 2. adımına geçerek yolların tutulduğu bu matrisi adım adım güncelleyelim:

	A	B	C	D	E
A	0	10	∞	5	∞
B	10	0	5	5	10
C	∞	5	0	∞	∞
D	5	5	∞	0	20
E	∞	10	∞	20	0

B üzerinden atlanarak ulaşılan düğümleri güncelleyelim

	A	B	C	D	E
A	0	10	**15**	5	**20**
B	10	0	5	5	10
C	**15**	5	0	**10**	**15**
D	5	5	**10**	0	**15**
E	**20**	10	**15**	**15**	0

C üzerinden atlanan düğümler:

	A	B	C	D	E
A	0	10	15	5	20
B	10	0	5	5	10
C	15	5	0	10	15
D	5	5	10	0	15
E	20	10	15	15	0

D üzerinden atlanan düğümler:

	A	B	C	D	E
A	0	10	15	5	20
B	10	0	5	5	10
C	15	5	0	10	15

```
D   5   5  10   0  15
E  20  10  15  15   0
```

E üzerinden atlanan düğümler:

```
    A   B   C   D   E
A   0  10  15   5  20
B  10   0   5   5  10
C  15   5   0  10  15
D   5   5  10   0  15
E  20  10  15  15   0
```

Yukarıda son elde edilen bu matriste görüldüğü üzere herhangi bir düğümden diğer bütün düğümlere giden en kısa yollar çıkarılmıştır. Örneğin A düğümünden E'ye 20 uzunluğunda veya C düğümünden D'ye 10 uzunluğunda yolla gidilebilir.

Yukarıdaki matrislerde diyagona göre simetri bulunmasının sebebi grafın yönsüz graf (undirected graph) olmasıdır. Şayet graf yönlü graf (directed graph) olsaydı bu simetri bozulurdu (tabi yönlerin ağırlıklarının aynı olmaması durumunda).

Algoritmanın kullanım alanları

Floyd-Warshall algoritması aşağıdaki amaçlar için kullanılabilir:

- Yönlü graflarda en kısa yolun bulunması için. (Yukarıda bu durumu gösteren bir örnek bulunmakta)
- Bir düğümden seyahat edilebilecek diğer düğümlerin bulunmasında (transitive closure). Örneğin matrisin ilk halinde gidilemeyen düğümler sonsuz değerine sahiptir. Şayet matris işlendikten sonra hala sonsuz değerine sahip matris elemanı bulunursa bunun anlamı o satır ve sütündaki düğümler arasında gidişin aktarmalı da olsa mümkün olmadığıdır.
- Düzenli ifadelerde (regular expressions) herhangi bir tekrarlı olayın tespiti için kullanılabilir. Kleen yıldızı (kleen's algorithm) şeklinde tekrar eden ifadelerin bulunmasına yarar. Şayet çıkan matriste bir düğümden diğer düğüme giden yol sonsuz değilse ve diyagona göre simetriği olan yol da sonsuz değilse bir kleen yıldızı üretilebilir.
- Ağ programlamasında özellikle yönlendirici algoritmalarında (routing algorithms) en kısa yolun tayin edilmesinde kullanılabilir.
- Yönsüz bir grafın (undirected graph), iki parçalı graf (bipartite graph) olup olmadığının bulunmasında kullanılabilir. Basitçe graftaki mesafelerin tamamını 1 uzunluğunda kabul edersek (veya mesafe olarak atlanan düğüm sayısını (hop count) kabul edersek) bu durumda bir düğümden gidilebilen bütün komşu düğümlerin mesafesi ya tek ya da çift olmalıdır. Bir düğümün hem tek hem de çift komşusu varsa bipartite değildir denilebilir.

SORU 32: Eşleme (Matching)

Bilgisayar bilimlerinde çeşitli amaçlar için kullanılan eşleştirme problemlerinin genel ismidir. Genellikle bir arz ile bir talebin eşleştirilmesi şeklinde olur. Örneğin bilgisayarın kaynaklarının, bu kaynakları talep eden işlemler ile eşleştirilmesi gibi. Ya da gerçek hayattan

bir çalışanın uygun iş ile eşleştirilmesi veya evlilik problemleri veya iş akış diyagramları (işin doğru kaynak üzerinden akması) gibi problemler bu grupta sayılabilir.

İçerik

Eşleme problemleri genel olarak bir graf problemi olarak da düşünülebilir. Yani amaçlanan eşleme işlemi bir graf ile modellenebilir ve bu model üzerinde graf teorsinin bize sunduğu bütün imkanlar kullanılabilir.

Örneğin Petri Ağları (Petri Networks) bu konuda oldukça uygun bir çözüm ortamıdır. Benzer şekilde koşul programlama (constraint programming) başlığı altında da pek çok graf modellemesi ve çözümü (örneğin kiriş koşulu (arc constraint)) kullanmak mümkündür.

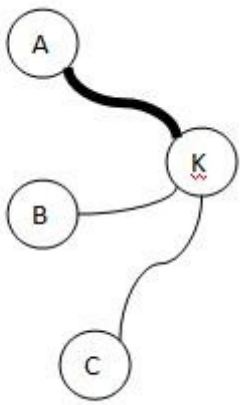
Eşleme problemlerinde genellikle kenar bağımsız kümesi (edge independent set) bulunmaya çalışılır. Bu küme genellikle eşleşecek olan varlıkları (grafta genellikle düğümler (nodes) ile gösterilir) eşlenmiş olarak modeller ve eşlenmeyen dışarıda kalan düğümlerin tespitini kolaylaştırır.

1. Eşleşme Tipleri

Bir grafta bulunan bir düğüm (node , vertex) şayet bir kenara (edge) bağlıysa bu düğüm eşleşmiş kabul edilir (matched). Şayet bir kenara bağlı değilse, eşleşmemiş (unmatched) kabul edilir.

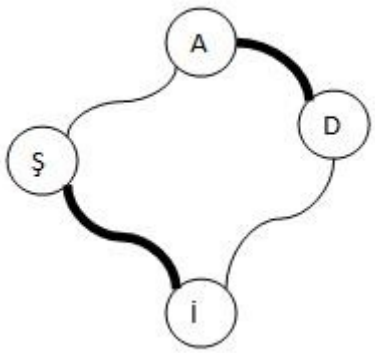
a. Çoklu Eşleşme (Maximal Matching)

Bir grafta ortak düğümleri bulunmaksızın alınabilecek en fazla kenar sayısı o grafın çoklu eşlemesini veriri (maximal match). Örneğin aşağıda bu duruma örnek graflar bulunmaktadır:



Yukarıdaki grafta 3 kenar (edge) bulunmaktadır ve çoklu eşleşme bu kenarlardan sadece birisinin alınması ile olabilir. YAni A-K kenarı alındığı için B-K veya C-K kenarları alınamaz çünkü bu kenarlardan herhangi birisinin daha alınması durumunda K düğümü ile ortak kesişim düğümü bulunmuş olacaktır.

Alternatif olarak yukarıdaki grafta sadece B-K veya sadece C-K düğümleri alınabilir. Ancak yukarıdaki çoklu eşleme sonucunda 1 kenar alınabilmektedir.

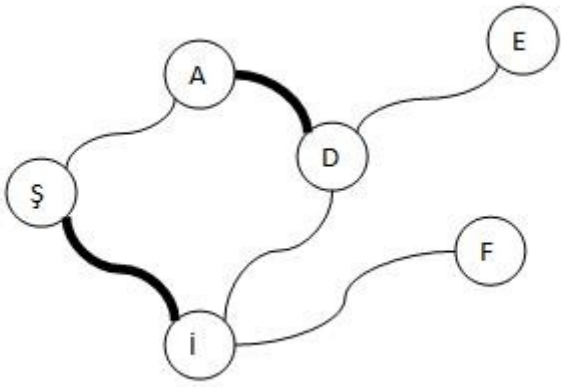


Örneğin yukarıdaki grafta çoklu eşleşme sonucunda iki kenar alınabilmektedir. Yukarıdaki grafta bu kenarlar Ş-İ ve A-D olarak belirlenmiştir. Alternatif olarak Ş-A ve D-İ kenarları da olabilirdi ancak örneğin Ş-A ve A-D kenarları aynı anda alınamaz çünkü bu durumda A düğümü hem Ş hem de D düğümü ile eşleşmiş olur.

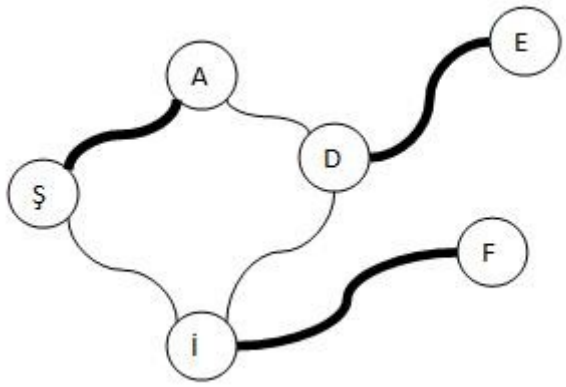
Amacımız olan eşleşmeye geri dönecek ve bu bilgiyi yorumlayacak olursak. Bir düğüm (bir kaynak, kişi ya da varlık) sadece bir düğüm ile eşleşebilir. Örneğin bir kişi sadece bir başka kişi ile evlenebilir veya bir kişinin sadece bir işi olabilir şeklinde düşünülebilir. Bu durumun dışındaki talepler graf teorisinde farklı şekillerde çözülür. Şimdilik bu eşleşme tanımı üzerinden farklı eşleşme türlerini tanıyalım.

b. Azami Eşleme (Maximum Matching)

Azami eşleme, çoklu eşlemeden farklı olarak bir grafta bulunabilecek en fazla eşi arar. Örneğin aşağıdaki şekilde bu iki eşleme arasındaki farkı görebiliriz.



Örneğin yukarıdaki şekilde birbiri ile komşu olmadığı için alınan iki kenar (edge) görülüyor. Bu durumda yukarıdaki graf çoklu eşleme (maximal matching) için uygun bir graf iken azami eşeleme (maximum matching) kabul edilemez. Yukarıdaki grafın azami eşleme olması için aşağıdaki şekilde alınabilecek en fazla kenarı eşlemesi gerekir:

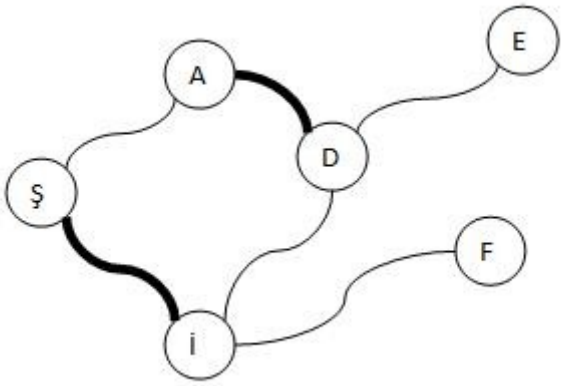


Yukarıdaki şekilde, bir önceki şekilden farklı olarak iki kenar yerine 3 kenar alınmıştır. Yukaırdaki şekilde 4 kenar eşlemenin imkanı bulunmadığı için azami eşleme (maximum matching) olarak kabul edilebilir.

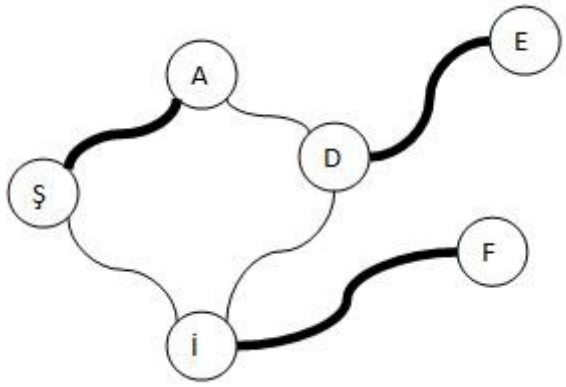
c. Mükemmel Eşleme (Perfect Matching)

Şayet bir graftaki bütün düğümler bir başka düğüm ile eşlendiyse ve eşleşmemiş bir düğüm kalmadıysa bu eşleme tipine mükemmel eşleme ismi verilir.

Örneğin aşağıdaki grafta yapılan eşleme mükemmel eşleme değildir:



Yukarıdaki graf mükemmel eşleme değildir çünkü E ve F düğümleri eşleme dışında bırakılmıştır.



Buna karşılık yukarıdaki eşleme mükemmel eşleme olarak kabul edilebilir çünkü bütün düğümler en az bir eşlemenin içine alınmıştır.

d. Yaklaşık Mükemmel Eşleme (Near Perfect Matching)

Bu eşleme, mükemmel eşlemeden farklı olarak tek bir düğümün eşlenmemesi durumunu kabul eder. Yani grafta şayet tek sayıda düğüm varsa eşleme sonucunda bir düğümün dışarıda kalması zaten beklenen bir durumdur. İşet bu durumda diğer bütün düğümler eşleniyor ancak tek bir düğüm dışarıda kalıyorsa buna yaklaşık mükemmel eşleme ismi verilir.

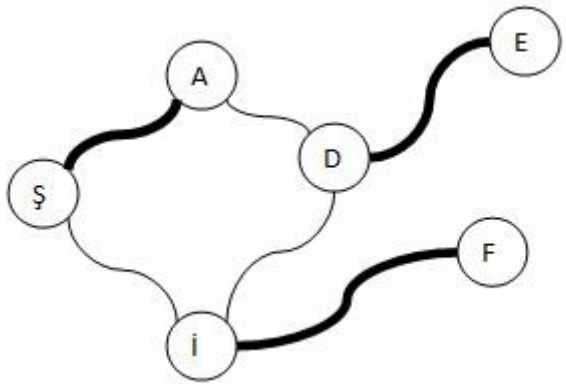
2. Eşleme sonucu yollar (Paths)

Bir eşleme (matching) işlemi sonucunda grafta elde edilen yollar için aşağıdaki tanımlar yapılabilir.

a. Dalgalı Yol (Alternating Path)

Şayet grafta elde edilen yol bir eşlenmiş bir de eşlenmemiş şeklinde devam ediyorsa bu yola dalgalı yol ismi verilir.

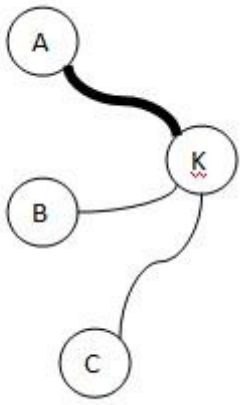
Örneğin aşağıdaki grafta bu durum görülebilir:



Yukarıdaki grafta F-İ-Ş-A-D-E yolu veya F-İ-D-E yolu birer dalgalı yoldur çünkü yoldaki eşlemeler sonucunda geçilen kenarlar bir eşlenmiş bir de eşlenmemiş olarak sınıflandırlabilir.

b. Uzatılmış Yol (Augmented Path)

Bir dalgalı yolun ilave bir düğüm ile başlaması durumudur. Yani şayet yola eşleşmemiş bir düğüm ile başlanıyorsa yada farklı bir ifadeyle başlanan düğüm herhangi bir eşleme içerisinde değilse bu yola uzatılmış yol ismi verilir.



Örneğin yukarıdaki şekilde C-K-A veya B-K-A yolları uzatılmış yollara örnektir. Başlana düğümler herhangi bir eşleme içerisinde değildir.

SORU 33: Petri Ağları (Petri Nets)

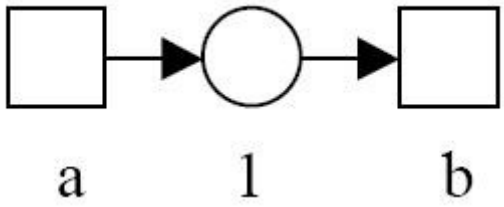
Bilgisayar bilimlerinde özellikle birbiri ile eş zamanda çalışan işlerin (concurrent jobs) modellenmesi ve çözülmesinde kullanılan özel grafiklerdir. Bu graflara Yer / Geçiş Ağları (Place / Transition Networks veya P/T Nets) ismi de verilir.

İçerik

Özellikle dağıtık sistemlerde (distributed systems), paralel programlamada (prallel programming) ve eş zamanlı çalışmada (concurrent processing) oldukça sık kullanılan bu ağlar yönlü iki parçalı ağaç (directed bipartite graph) olarak sınıflandırılabilir. Buna göre petri ağlarında (petri nets) bir olay ya da geçiş (transition) bir yer yada koşul (place) ve bir yön (ok, arc) bulunur. Örneğin iş akışını gösteren bir diyagramda bir işin bir noktadan başlayarak sonuca kadar izlediği rota çizilirken geçtiği yerler ve geçerken bazı koşullara göre farklı yerlere yönlendirilmesinden söz edilebilir. İşte iş akış grafikleri klasik birer petri ağıdır.

Örnek Petri Ağları

Örneğin aşağıda basit bir petri ağı gösterilmektedir:



Bir petri ağında daireler yerleri (places) ve kareler geçişleri (transitions) temsil eder. Yukarıdaki ağda a ve b birer geçiş (transition) iken 1 bir yerdir (place). Oklar ise yerler ve geçiler arasındaki akışı gösterir. Bu geçişleri birer zaman akışı olarak düşünmek de mümkündür. Örneğin iş akış şemasında bir işin akışının zamanda ilerlemesi gösterilir.

Yukarıdaki örnek petri ağı basit bir yaşamı göstermek için çizilmiştir. Bu durumda

a: doğum

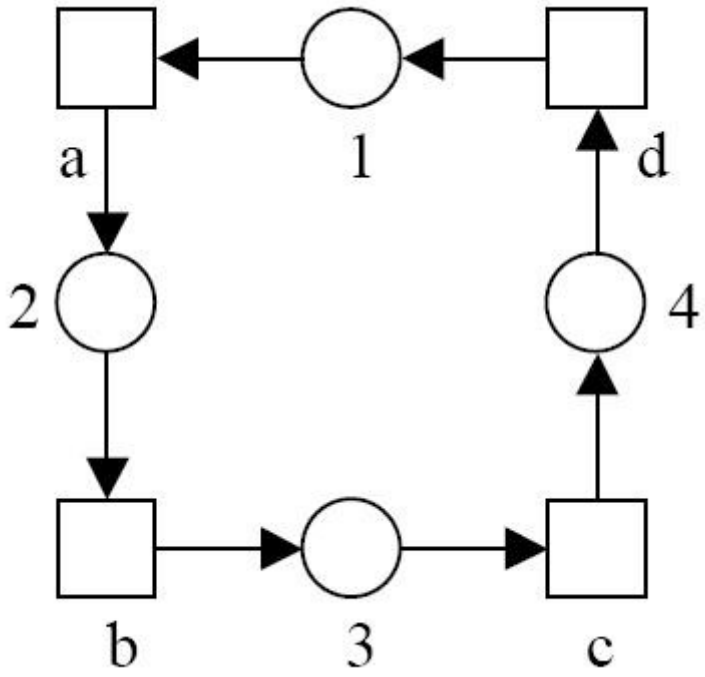
1: hayatın yaşanması

b: ölüm

olarak yorumlanabilir. Yani bir canlının basit bir şekilde doğması yaşaması ve ölmesi olarak düşünülebilir.

Döngüsel Petri Ağları (Circular Petri Networks)

Benzer bir örneği mevsimlerden verebiliriz. Örneğin dört mevsim sürekli olarak birbirini kovalar ve ayrıca mevsimlerin geçişi söz konusudur.

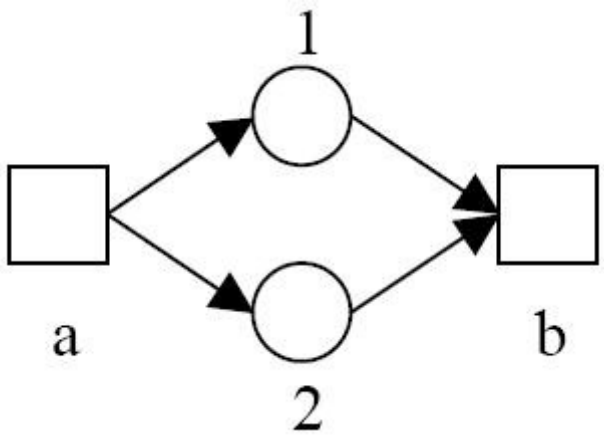


Örneğin yukarıdaki petri ağında 1′den 4′e kadar olan daireler mevsimleri (yaz, bahar, kış, güz) ve a-d arasındaki harflerde mevsim dönüşlerini (ekinoks) göstermektedir.

Yukarıdaki şekilde bir döngü halinde mevsim dönüşü görülmektedir.

Paralel Petri Ağları

Petri ağlarının eş zamanlı (paralel) işlemler için kullanıldığını görmüştük. Şimdiye kadar olan örneklerde hep birbirini izleyen işler modellendi. Şimdi paralel olayların nasıl modellendiğine bakalım:



Örneğin yukarıdaki şekil iki paralel olayı (1 ve 2) modellemek için kullanılmıştır. Yukarıda 2007-2009 yılları arasında öğrenci olan ve aynı zamanda çalışan bir kişinin aynı anda yaptığı bu işler modellenmiş olsun.

Bu durumda a: 2007 , b ise 2009′u göstermektedir. Yani olayların başlangıç ve bitişleri gösterilir.

1 numara ile okuma eylemi ve 2 numara ile çalışma eylemi modellenmiş kabul edilirse bu iki eylemin başlangıcı ve bitişi aynı tarihlere tesadüf etmiştir.

Dolayısıyla yukarıdaki şekilden iki farklı eylemin aynı başlangıç ve bitişle paralel olduğunu söyleyebiliriz.

Koşullu Petri Ağları

Petri ağlarında ayrıca koşul bulundurmak ve bu koşula göre zamanın akışını şekillendirmek mümkündür.

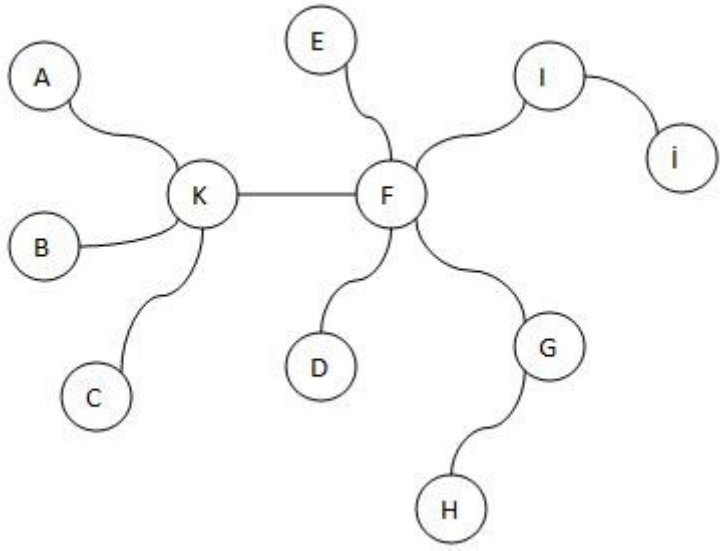
SORU 34: İki Parçalı Graflar (Bipartite Graphs)

İçerik

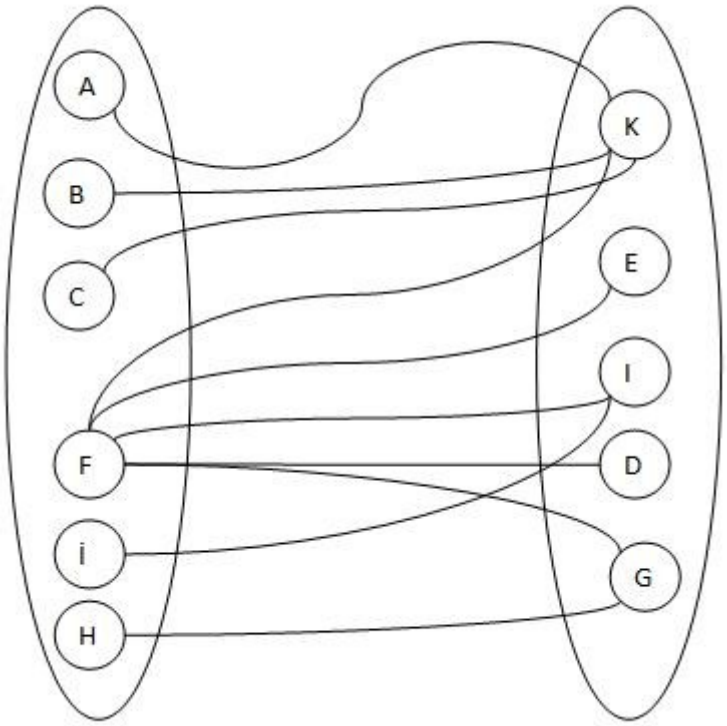
Bilgisayar bilimlerinde veri modellemede sıkça kullanılan grafların (graph) özel bir durumudur. Buna göre bir graf'ı oluşturan düğümleri iki farklı kümeye ayırabiliyorsak ve bu iki kümenin elemanlarından küme içerisindeki bir elemana gidilmiyorsa. Yani bütün kenarlar (edges) kümeler arasındaki elemanlar arasındaysa, bu graflara iki parçalı graf (bipartite graph) ismi verilir.

İki parçalı graflara örnekler

Örneğin aşağıdaki şekilde gösterilen graf'ın iki parçalı olup olamayacağını incelemeye çalışalım:

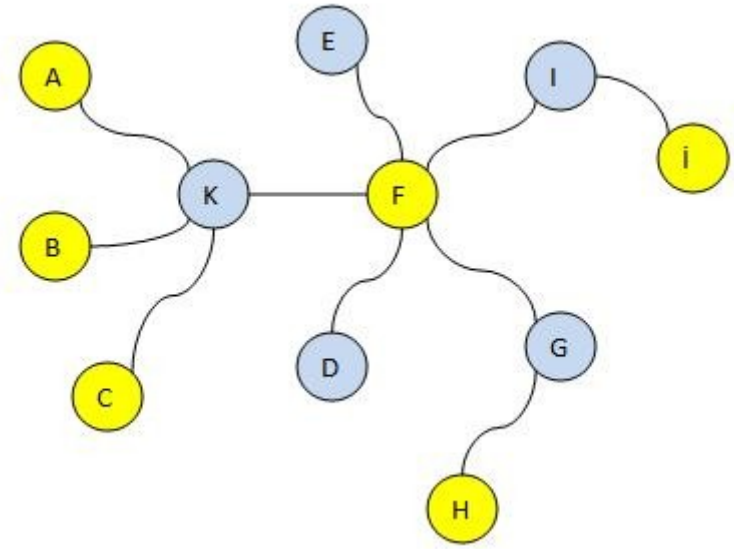


Yukarıda örnek olarak verilen garfın düğümlerini koşulumuzu sağlayacak şekilde yani iki ayrı küme oluşturacak ve bu kümelerin içerisinde yol bulunmayacak şekilde yerleştirmeye çalışalım.



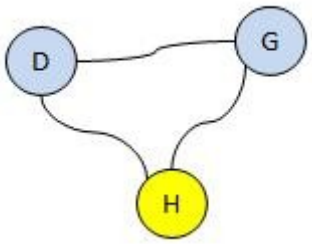
Yukarıdaki şekilde, ilk şekilden farklı olarak sadece düğümlerin (nodes) yerleri değiştirilmiştir. Yukarıdaki şekilde ayrıca iki ayrı küme oluşturulmuş ve ilk küme ile ikinci küme arasında bölünen düğümlerin tamamında kenarlar karşı kümeye işaret etmiştir. Yani aynı küme içerisinde hiçbir kenar bulunmamaktadır.

Bu durumu aşağıdaki şekilde de gösterebilirdik:

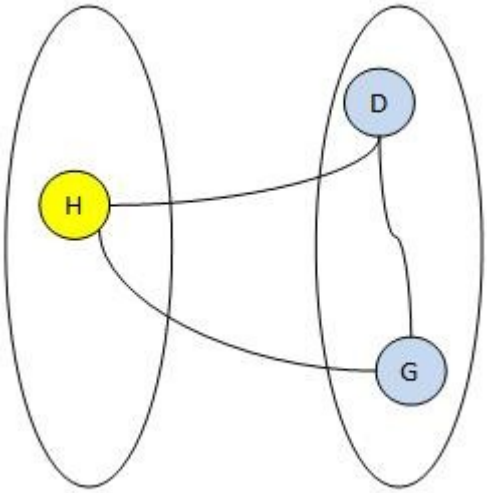


Yukarıdaki şekilde görüldüğü üzere komşu olmayan düğümler renklendirilmiştir. Yani komşu iki düğüm aynı renkle renklendirilmemiştir. Daha farklı bir deyimle bir düğümün komşuluk listesindeki (adjacency list) bütün düğümler aynı renkte boyanmıştır. Sonuçta aynı renkte boyanan iki komşu oluşmamaktadır.

Aşağıdaki iki parçalı olmayan grafta aynı şekilde yaklaşırsak problem orataya çıkacaktır.



Yukarıdaki şeki bipartite tree (iki parçalı ağaç) değildir. Çünkü graftaki 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:

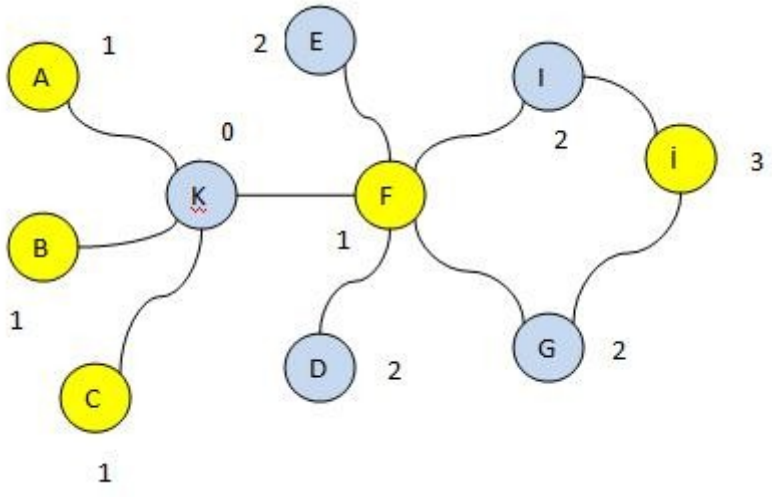


Grafı yukarıdaki şekilde iki gruba bölmek istersek görüldüğü üzere aynı gruptaki iki düğüm arasında bir kenar bulunmakta bu durumda da iki parçalı ağaç olamamaktadır.

İki parçalı grafın test edilmesi

Bir grafın iki parçalı olup olmadığını (bipartite) test etmek için ne yazık ki yukarıda gösterildiği gibi graftaki bütün düğümlerin yerlerinin değiştirilmesi iki kümede toplanması ve sonunda da aralarında bir kenar olup olmadığına bakılması mümkün değildir.

Bir grafın iki parçalı olduğunu anlamak için grafın 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 grafa iki parçalı graf ismi verilebilir.



Yukarıdaki şekilde başlangıç için rast gele olarak 0 seçilmiş olsun. Bu grafta 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 grafların 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 grafların özellikleri

iki parçalı graflar için aşağıdaki çıkarımlar ve koşullar sıralanabilir:

- Bütün ağaçlar iki parçalı graftır.
- Şayet bir graf döngü (cycle) içermiyorsa iki parçalı graftır.
- Şayet bir graf tek sayıda düğümden oluşan bir döngü (cycle) içeriyorsa iki parçalı graf değildir.
- Şayet bir graf boyandığında iki renk veya daha az renkle boyana biliyorsa bu graf iki parçalı graftır. (detaylı bilgi için graf boyama (graph coloring) başlıklı yazıyı okuyabilirsiniz)

SORU 35: Minimax Ağaçları (Minimax Tree)

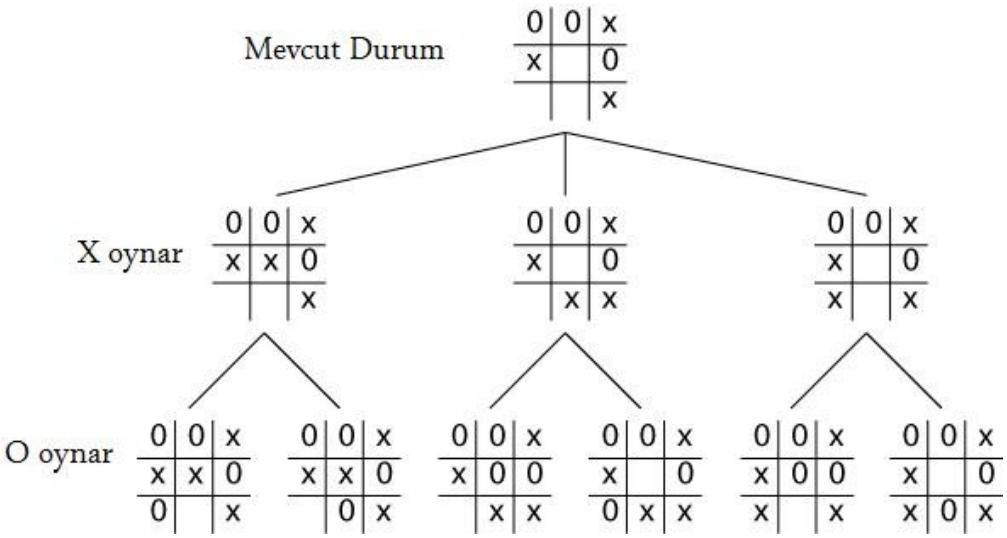
Bilgisayar mühendisliğinde, yapay zeka konusunda kullanılan bir karar ağacı türüdür. Aslında minimax ağaçları bilgisayar bilimlerine işletme bilimindeki oyun teorisinden (game theory) girmiştir.

Temel olarak sıfır toplamlı bir oyunda (zero sum game), yani birisinin kaybının başka birisinin kazancı olduğu (veya tam tersi) oyunlarda karar vermek için kullanılışlıdırlar. Örneğin çoğu masa oyunu (satranç, othello, tictactoe gibi) veya çoğu finansal oyunlar (borsa gibi) veya çoğu kumar oyunları sıfır toplamlı oyunlar arasında sayılabilir (yani birisinin kaybı başka birisinin kazancıdır ve sonuçta toplam sıfır olur).

Yukarıda bahsedilen bu oyunlarda doğru karar verilmesini sağlayan minimax ağacı basitçe kaybı asgariye indirmeye (mimize etmeye) ve dolayısıyla kazancı azamiye çıkarmaya (maximize etmeye) çalışır.

Ağaç basitçe her düğümde (node) farklı alternatiflerin değerlerini hesaplar. Son düğümden (yapraklardan ,leaf) yukarıya doğru değerleri seçerek gelir ve en sonunda bütün ağaçtaki en doğru seçenek seçilmiş olur.

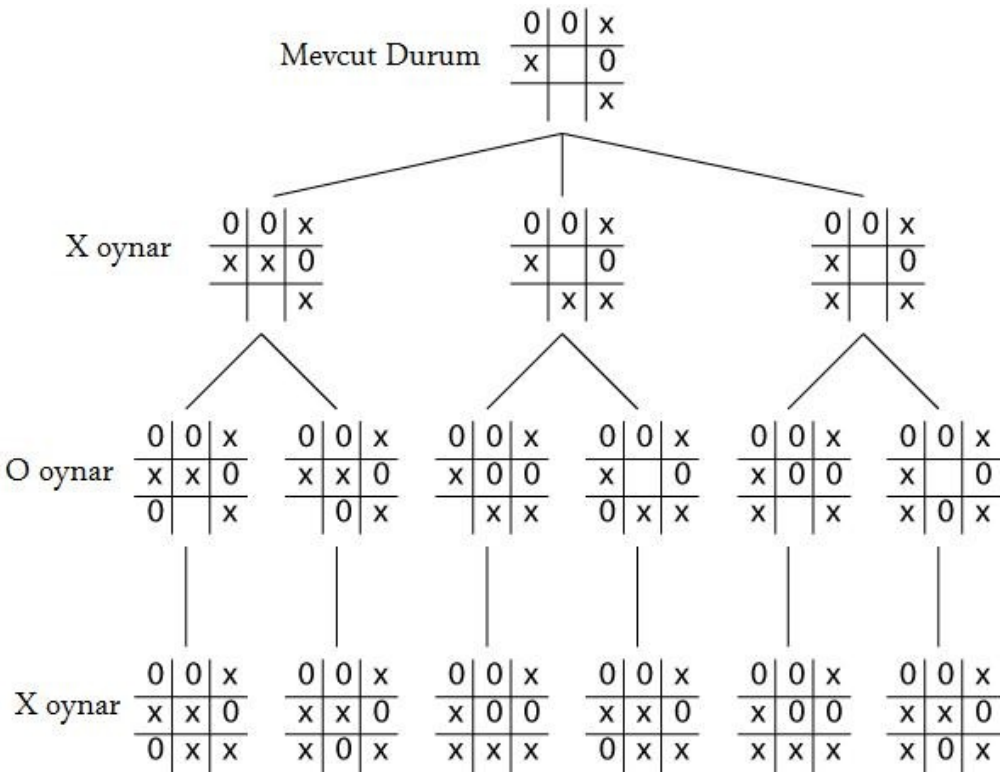
Örneğin Tic Tac Toe oyununu ele alalım. 3×3 boyutundaki kare bir tabloda sırasıyla X ve O harflerinin taraflarca yazıldığı bu oyunda, oyunun durumuna göre aşağıdakine benzer bir karar ağacı çıkması mümkündür:



Yukarıda ağacın kökünü (root) oluşturan ilk durumda oyunun mevcut hali görülmektedir. Ardından karar verecek taraf olan ve tahtaya X sembolü koyan oyuncu kendi oyununu analiz eder. X 'in oynanabileceği 3 ayrı ihtimal bulunmaktadır ve bütün bu ihtimaller ağaçta farklı birer alt düğüm (node) olarak gösterilmiştir.

Ağacın daha alt seviyesinde ise X'in oyananabileceği her ihtimalde, O'nun oyananabileceği ihtimaller gösterilmiştir.

Bu durumda X oynayacak olan oyuncu bütün ihtimalleri görmüş olur ve ilave olarak kendisinin yapacağı son hamleleri aşağıdaki şekilde hesaplar:



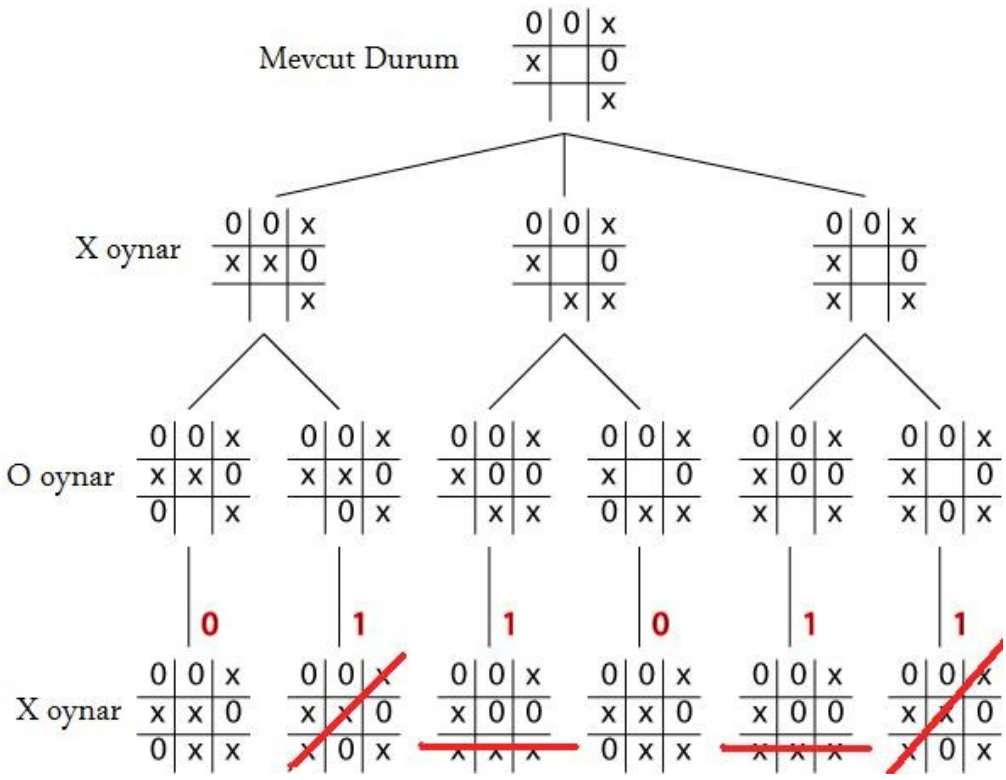
Yukarıdaki şekile X'in yapacağı son hamleninde hesaplanışını görüyoruz. Bir minimax ağacının hesaplanması sırasında kaç seviye gidileceğine seviye (ply) ismi verilir. Örneğin yukarıdaki ilk şekildeki minimax ağacımı 2 seviye (ply) bir ağaçken, yukarıdaki son ağacımız 3 seviyeli bir ağaçtır.

Minimax ağacının seviyesinin artması sonucun daha kesin bulunması ve yapay zekamızın başarısını arttırır. Ancak bunun karşılığında bilgisayarın hafızasında (RAM, memory) daha fazla bilginin tutulması gibi bir bedel ödenir. Örneğin satranç veya GO gibi ihtimallerin çok yüksek olduğu oyunlarda seviyenin belirli bir limitin altında tutulması gerekir.

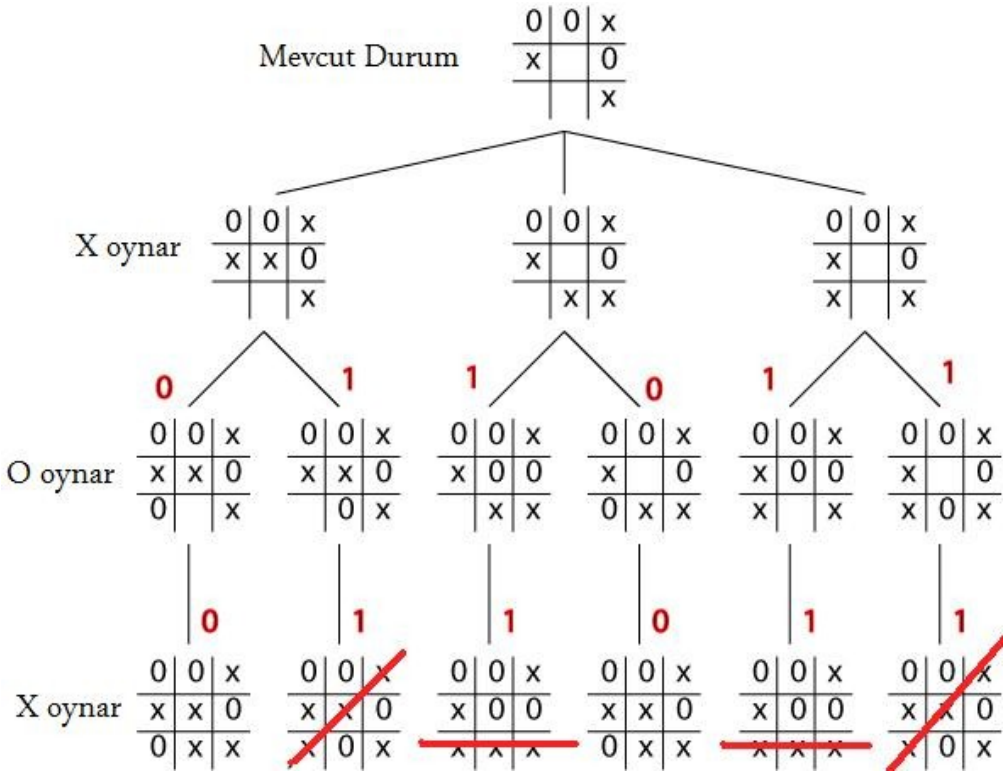
Bilgisayarın yukarıdaki ağaca bakarak bir karar vermesi çok güçtür. Kararın sayısal bir değere oturması için yukarıdaki her hamle durumunu puanlamamız ve bundan sonra bilgisayarın minimax yaklaşımı ile seçim yapmasını istememiz gerekir.

Örneğin bilgisayarın (Burada X olarak oynadığını düşünelim) kazanma durumlarına 1 ve kaybetme durumlarına 0 diyelim.

Bu durumda ihtimaller ve puanlar aşağıdaki şekilde olacaktır:



Yukarıdaki şekilden de anlaşılacağı üzere bilgisayar kazancının en fazla olduğu ihtimali seçmek ister. Basit bir hesapla bilgisayarın kazanç durumlarının toplamlarını ağacın bir üst seviyesine taşıyalım:



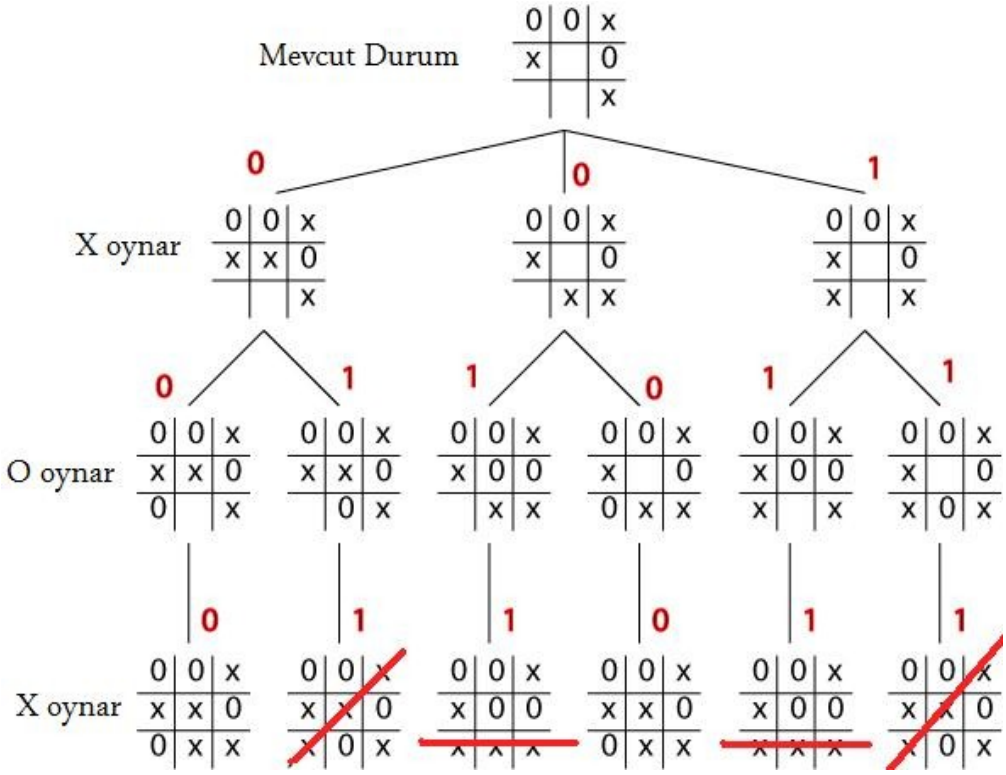
Yukarıda ağacın son seviyesinden karar vermemiz gereken pozisyona doğru bir seviye puanlar taşınmıştır. Yani kararın verilmesi gereken pozisyon ağacın kökündeki (en üst

seviyesindeki) pozisyondur ve oyun sonu en altta gösterilmektedir. İşte bizde puanlarımızı karara bir seviye yaklaştırıyoruz.

Bu yaklaştırma işlemi sırasında dikkat edilmesi gereken X'in hamle sırasındaki azami (maximum) değerlerin taşınmasıdır çünkü X kendi hamlesinin en çok puan getirdiği durumu almak ister. Ancak yukarıdaki grafikteki son seviyede X'in tek hamlesi olduğu için mecburen (bir seçim yapılmaksızın) puanlar bir seviye yukarı taşınmıştır.

O'nun hamlesinin bulunduğu bir üst seviyede ise bir seçim yapılabilir. Çünkü O hamlesini iki farklı yere yapabilir. Bu durumda O'nun kendisi için en avantajlı yere oynayacağını düşünürsek bu seviyede asgari (minimum) değerlerin alınması gerekir.

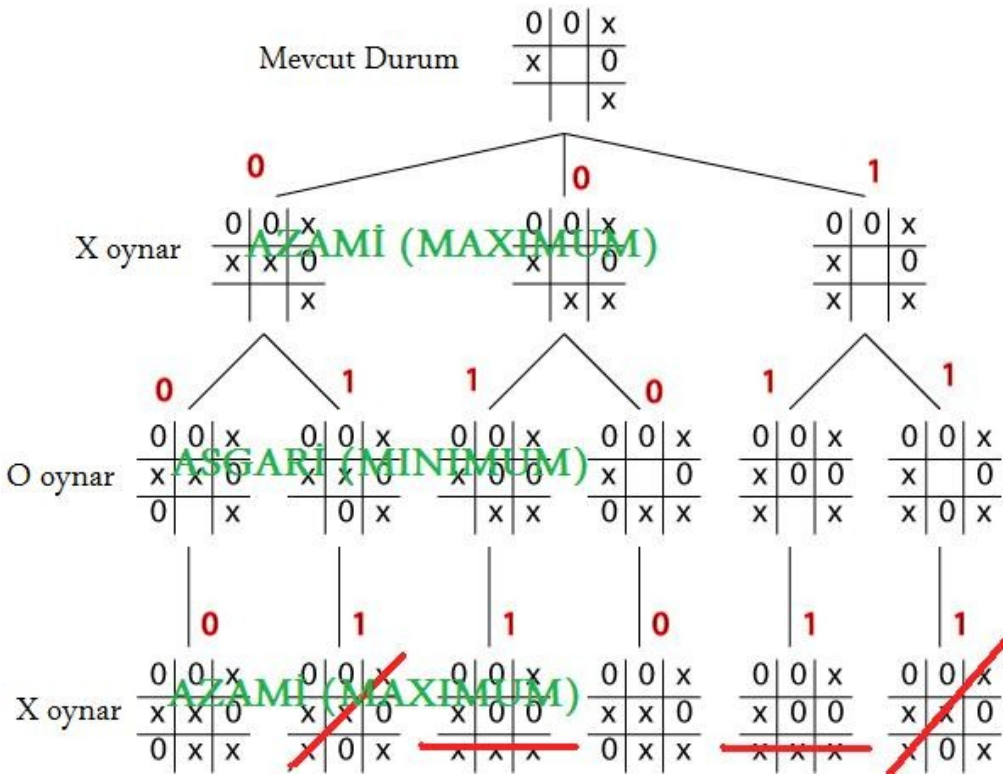
Aşağıda O'nun hamlesi için alınmış asgari değerler bulunuyor :



Görüldüğü gibi ihtimallerin en düşük değerleri alınmıştır.

Taşınan bu son seviyeyle artık bilgisayar hamlesine karar verebilir çünkü mevcut durum için hangi hamlenin en kârlı olduğu (max) görülmektedir. Yukarıdaki ağaçta en sağdaki değer olan 1 puanındaki hamleyi yapması durumunda bilgisayar her ihtimalde oyunu kazanmaktadır.

Sonuç olarak alınan değerler aşağıdaki şekilde gösterilebilir:



Yani bir min bir max alınmaktadır ve bu işlem bu şekilde devam etmektedir. Ağacımızın ismi de zaten buradan türetilmiş minimax ağacı olarak geçmektedir. Aynı anlamda farklı başlangıç değerinde maximin ağacı da literatürde geçmektedir. Bu ağaç da tamamen aynıdır ancak kaygı kendi hamleleri üzerine değil rakibin hamleleri üzerine kuruludur.

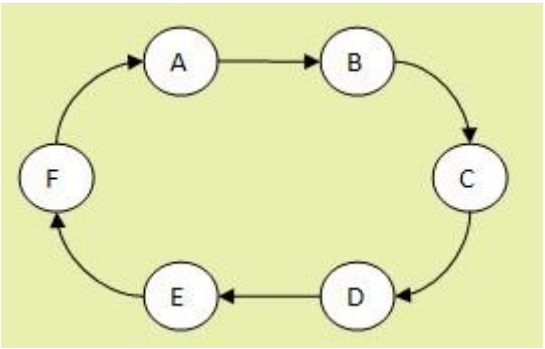
SORU 36: Brent Algoritması (Brent's Algorithm)

Bilgisayar bilimlerinde özlelikle graf teorisinde (graph theory) kullanılan ve bir döngüyü (cycle) algılamaya yarayan algoritmadır. (cycle detection).

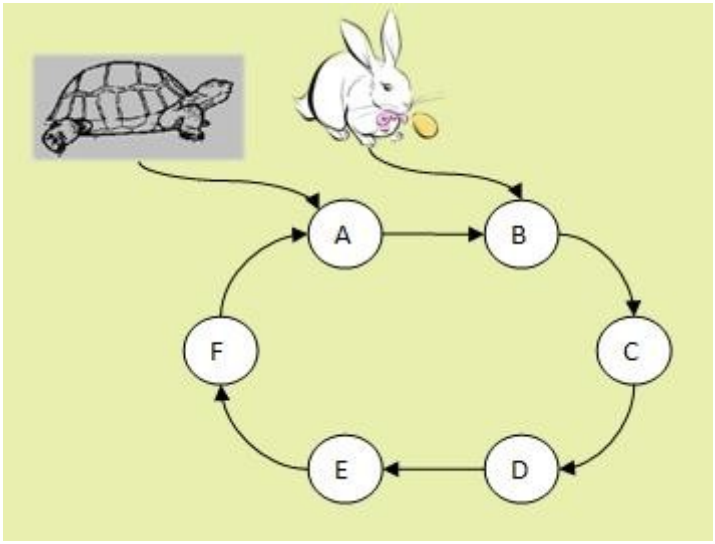
Basitçe tavşan ve kaplumbağa algoritmasından (hare and tortoise algoritm) esinlenmiştir. Floyd algoritması olarak da isimlendirilen tavşan ve kaplumbağa algoritmasından farklı olarak tavşan, kaplumbağanın iki misli değil 2 üzeri adımla hareket etmektedir.

Yani kaplumbağa, tavşan 2′nin bir kuvveti olan adım attığında hareket etmekte, bu zamana kadar beklemektedir.

Örneğin aşağıdaki dairenin Brent algoritması ile nasıl çözüldüğünü görmeye çalışalım:



Başlangıç değeri olarak A düğümünden iki göstericinin (pointer) başladığını kabul edelim. Bu durumda başlangıçtaki tavşan ve kaplumbağanın şekli aşağıdaki gibi olacaktır :



Kaplumbağa tavşanı beklemektedir. Tavşan birer birer hareket etmekte, şayet hareket ettiği değer 2 üzeri bir değer olursa kaplumbağa da hareket etmektedir. Aynı düğüme geldiklerinde daire bulunmuş demektir.

Yukarıdaki şekil için tavşan ve kaplumbağanın dolaştıkları düğümleri sıralayacak olursak:

K T

—

A B

A C (C tavşanın geldiği 2. düğüm olduğu için kaplumbağa hareket eder)

B D

B E (E tavşanın 4. adımda geldiği düğüm olduğu ve ikinin kuvveti olduğu için kaplumbağa hareket eder)

C F

C A

C B

C C (C. tavşanın 8. adımda geldiği düğüm olduğu için kaplumbağa hareket edecektir ancak daire bulunmuştur)

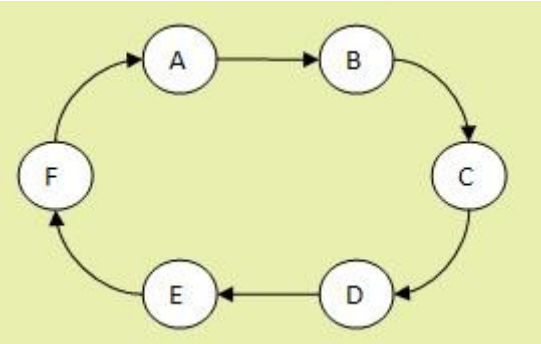
Yukarıdaki örnekte iki gösterici de C düğümünde buluşmaktadır dolayısıyla daire yakalanmıştır.

SORU 37: Tavşan Kaplumbağa Algoritması (Hare and Tortoise Algorithm)

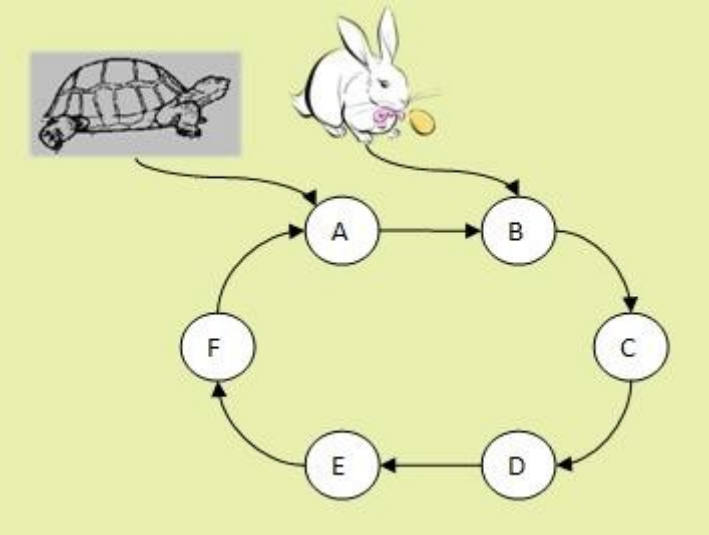
Bilgisayar bilimlerinde veriyi modellemek için kullanılan graflarda bir döngü (cycle) olup olmadığını algılamaya yaramak için kullanılan algoritmadır. Floyd Döngü Yakalama Algoritması (Floyd's Cycle Detection Algorithm) olarak da geçen bu algoritmaya göre bir yol üzerinde hareket eden iki farklı hızdaki gösterici (pointer) aynı değeri gösterebiliyorsa burada bir döngü bulunuyor demektir.

Tosbağa ve tavşan benzetmesinde ise tavşan, tobağadan iki misli hızlı gitmekte ve dolayısıyla bir zamandan şayet döngü bulunuyorsa mutlaka tosbağayı yakalaması beklenmektedir.

Örneğin aşağıdaki döngüyü ele alalım:



Yukarıdaki döngüyü kaplumbağa ve tavşanın dönüşünü inceleyelim. Başlangış düğümü olarak A'dan başlanıyor olsun.



Yukarıdaki örnekte kaplumbağa ve tavşanın başlangıç durumları ele alınmış. Tavşan kaplumbağadan iki misli hızlıl olduğu için daha ileride başlıyor ve birbirlerine eşit olana kadar sürekli hareket ediyorlar. Yani tavşan 2 kaplumbağa ise 1 hızında hareket ediyor.

Durumları aşağıdaki şekilde sıralanabilir:

```
T K
---
D B
```

```
F C
B D
D E
F F
```

Görüldüğü üzere tavşan her adımda ikişer atlarken kaplumbağa 1 adım ilerlemekte ve sonunda F düğümünde ikisi de aynı yeri göstermektedir. Buradan sonuç olarak bir döngü olduğuna varılabilir. Yani şayet 2 birim hızla giden ve 1 birim hızla giden iki gösterici (pointer) aynı yerde buluşuyorlarsa bunun anlamı ancak bir döngü olmasıdır.

Yukarıdaki graflar üzerinde kullanılan bu yaklaşım sayı teorisinde de (number theory) kullanılabilir. Örneğin dairesel grup (cyclic group) bulunması veya verilen bir sayı grubunun (dizisinin) dairesel olduğunun anlaşılması da mümkündür.

f(x) şeklinde dairesel olup olmadığı bilinmeyen bir fonksiyon düşünelim.

Kaplumbağa : f(x)

Tavşan : f(f(x))

şeklinde sayılar üretsinler. Bu durumda tavşan bir şekilde kaplumbağaya eşitse dairesel bir gruptan bahsediliyor demektir.

Örneğin

f(x) = 3x + 2 mod 7

olarak tanımlansın. Başlangıç değeri olarak

Kaplumbağa : f(1)

Tavşan : f(f(1))

değerleri ile başlıyor olsun. Bu durumda :

Kaplumbağa :5

Tavşan :3

değerini alacaktır.

Bulunan değerlerin tekrarlanması ile

Kaplumbağa : f(5)

Tavşan : f(f(3))

işlemini yapacak ve bu işlemlerin tekrarı aynı olana kadar devam edecektir.

```
K T
---
```

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

Yukarıda görüldüğü üzere kaplumbağa fonksiyonu bir kere işletirken tavşan iki kere işletmektedir. En nihayetinde aynı değerleri göstermesinden de anlaşılacağı üzere fonksiyonumuz dairesel bir fonksiyondur ve ürettiği sayıları tekrar eder.

Bu yaklaşım, hafızanın verimli kullanılmasını amaçlamaktadır. Yani bir dairenin yakalanması için geçilen bütün düğümler ve ya fonksiyon sonuçları bir yerde tutulup aynı değerden geçilip geçilmediğine bakılabilir. Ancak tahmin edileceği üzere bu işlem en kötü ihtimalde veri yapısında bulunan bütün sayıların ikinci bir yerde tutulması ile sonuçlanacak ve hafıza kullanımı artacaktır.

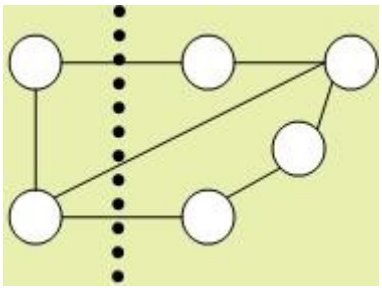
Tavşan ve kaplumbağa algoritmasında bunun tersine 2 gösterici (pointer) yeterlidir. Ancak bu sefer de işlem süresi uzamaktadır. Yukarıdaki örneklerde de görüleceği üzere aynı dairede birden fazla kere dönülmesi gerekebilmektedir.

Ayrıca karmaşık grafiklerde doğru yola karar verilmesi farklı bir problemdir. Örneğin bir ağaç üzerindeki dairenin yakalanması işleminde ağaçtaki yol ayrımlarının seçilmesi farklı bir problem halini alır.

SORU 38: Graflarda Kesitler (Cut in Graphs, Ağaçlarda Kesitler)

Graf teorisinde (graph theory) bir ağacın içerisinden bir kısmınkesilmesine dayalı işlemdir. Bir ağaç (tree) yada bir graf(graph) kesildiğinde tek bir kesilme ile iki parçaya ayrıldığı kabul edilir.

Bir kesitin boyutu kestiği kenar (edge) sayısı kadardır. Örneğin aşağıdaki kesitin boyutu 3′dür.

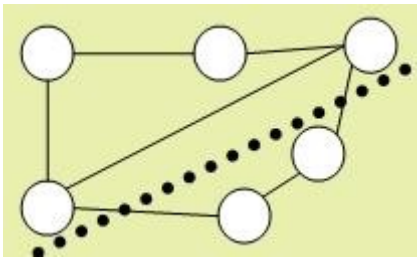


grafın ağırlıklı olması durumunda (weighted graph), kesitin boyutu, kestiği kenarların ağırlıkları topalmıdır.

Kesilme şekline göre bu ayrım iki grupta incelenebilir:

Asgari Kesit (Minimum Cut)

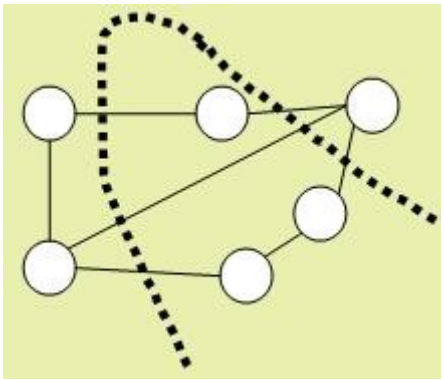
Bir grafı, iki parçaya bölen kesitin, en az kenarı kesmesi durumudur. Örneğin aşağıdaki grafta en az kesilebilen kenardan kesilmiştir. Bu kesilme dışındaki bütün kesitler bu kesitten daha yüksek veya aynı boyuttadır:



Yukarıdaki kesitin boyutu 2′dir ve yukarıdaki graf için boyutu 1 olan başka bir kesit bulunamaz.

Azami Kesit (Maximum Cut)

Kesitin, kesebildiği en fazla kenarı kesmesi durumudur. Aşağıdaki grafta bu durum gösterilmektedir:



Yukarıdaki kesitin boyutu 6′dır ve daha büyük boyutta bir kesit elde edilmesi mümkün değildir.

SORU 39: Komşuluk Listesi (Adjacency List)

Bir grafikteki her düğümün (node) komşularının listesine verilen isimdir. Örneğin aşağıdaki listeyi ele alalım:

Bu grafikte hangi düğümün hangi düğümlerle komşu olduğunu tutan birer liste çıkarılması mümkündür. Örneğin A düğümünün komşuluk listesi (adjacency list) {B,C,D} olurken D düğümünün komşuluk listesi : {A,C} olmaktadır. Bu durum yukarıdaki her düğüm için aşağıdaki tabloda gösterilmiştir:

Düğüm	Komşuluk Listesi
A	{ B,C,D }
B	{ A }
C	{ A,D }
D	{ A,C }

Yukarıdaki grafik ve tabloda dikkat edilecek noktalardan birisi de grafiğin yönsüz (undirected) olmasıdır. Grafiğin yönlü olması durumunda ulaşılabilecek yönün tayin edilmesi ve bu yönlerin komşuluk listesine alınması gerekir.

SORU 40: B Ağacı (B-Tree)

İçerik

İsminin nereden geldiği (B harfinin) tartışmalı olduğu bu ağaç yapısındaki amaç arama zamanını kısaltmaktır. Buna göre ağacın her düğümünde belirli sayıda anahtar veya kayıt tutularak arama işleminin hızlandırılması öngörülmüştür.

Arama hızının artmasına karşılık silme ve ekleme işlemlerinin nispeten yavaşlaması söz konusudur.

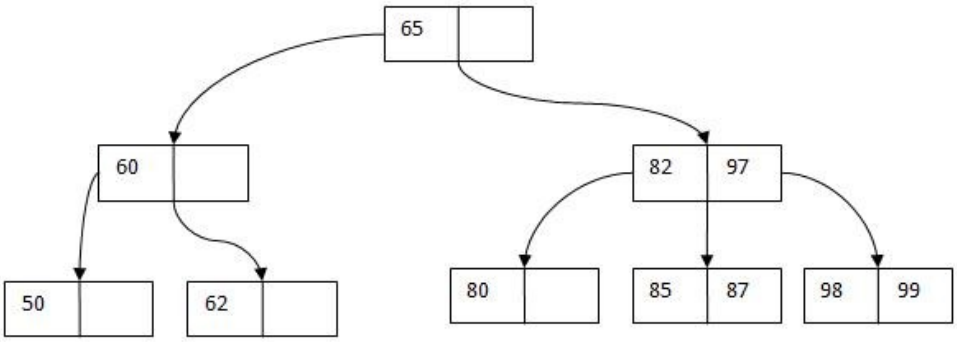
1. B-Ağacının tanımı

Bir B-Ağacı (B-Tree) aşağıdaki özelliklere sahip olmalıdır:

- Her düğümün (node) en fazla m çocuğu bulunmalıdır. (Bu sayının üzerinde eleman bulunursa düğümün çoğaltılması gerekir)
- Kök (root) ve yaprak (leaf) düğümleri haricindeki her düğümün en az m/2 adet elemanı bulunmalıdır. (Bu sayının altında eleman bulunursa düğüm kaldırılır)
- Bütün yapraklar aynı seviyede olmak zorundadır. Bir yaprağın seviyesinin düşmesi durumunda (daha yukarı çıkması veya daha sığ olması durumunda) ağaçta yapısal değişiklik gerekir.
- Herhangi bir düğümde k çocuk bulunuyorsa k-1 elemanı gösteren anahtar (key) bulunmalıdır.

2. Örnek B-Ağacı

Aşağıda örnek bir b ağacı gösterilmiştir:



Aşağıda b ağacı üzerinde yapılan ekleme, silme ve arama işlemleri açıklanmıştır.

3. B Ağacında Arama

Bağacında (Btree) arama işlemi kökten başlar. Aranan sayı kök düğümde bulunamaması halinde arama işlemi kökte bulunan anahtarların sağında solunda veya arasında şeklinde yönlendirilir. Örneğin yukarıdaki B-ağacında 87 anahtarı aranıyor olsun. Arama işlemi için aşağıdaki adımlar gerekir:

1. kök düğüme bakılır. 87 değeri 65′ten büyüktür. Kök düğümde tek anahtar olduğu için 65′in sağındaki gösterici(pointer) takip edilir.

2. 65. sağındaki düğüme gidilir ve ilk anahtar olan 82 ile aranan anahtar olan 87 karşılaştırılır. 87 değeri 82′den büyüktür. Öyleyse ikinci anahtar olan 97 ile karşılaştırılır. 87 bu değerden küçük olduğu için bu düğümde 82 ile 97 arasında bulunan gösterici izlenir.

3. Son olarak 82 ile 97 arasındaki düğüm izlenerek ulaşılan düğümdeki anahtar ile 87 karşılaştırılır. Bu düğümdeki ilk anahtar 85′tir. 87 bu değerden büyüktür. Düğümdeki bir sonraki anahtar alınır ve 87 değeri bulunur.

B-ağaçlarının bir özelliği ağacın her düğümündeki anahtarların sıralı oluşudur. Bu yüzden bir düğümde istenen anahtar aranırken, düğümde bulunan sayılara teker teker bakılır (linear search, doğrusal arama)

4. B Ağacına Ekleme

B ağaçlarında veri yaprak düğümlerden gösterilir (pointer). Yani aslında veri ağacın düğümlerinde değil son düğümlerden gösterilen hafıza bölmeleri (RAM veya Dosya) olarak tutulur. Dolayısıyla B ağacında ekleme işlemi sırasında anahtarlar üzerinden arama ve değişiklikler yapılır ve nihayetinde amaç B ağacının son düğümleri olan yaprak düğümlerden veriyi göstermektir.

B ağaçlarında veri eklemek için aşağıdaki adımlar takip edilir:

1. Ağaçta eklenecek doğru yaprak düğümü aranır. (Bu işlem için bir önceki adımda anlatılan arama algoritması kullanılır)

2. Şayet bulunan yaprak düğümde azami anahtar sayısından (maximum key number) daha az eleman varsa (yani anahtar eklemek için boş yer varsa) bu düğüme ekleme işlemi yapılır.

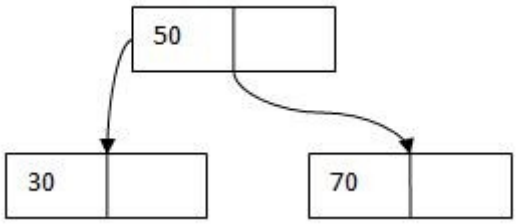
3. Şayet yeterli yer yoksa bu durumda bulunan bu yapra düğüm iki düğüme bölünür ve aşağıdaki adımlar izlenir:

1. Yeni eleman eklendikten sonra düğümde bulunan anahtarlar sıralanır ve ortadaki elemandan bölünür. (median değeri bulunur)

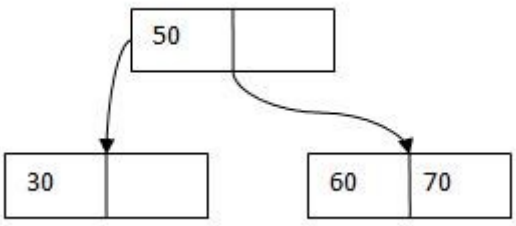
2. Ortanca değerden büyük elemanlar yeni oluşturulan sağ düğüme ve küçük elemanlar da sol düğüme konulur.

3. Ortanca eleman (median) ise bir üst düğüme konulur.

Yukarıdaki ekleme işlemini aşağıdaki örnek ağaç üzerinden görelim.

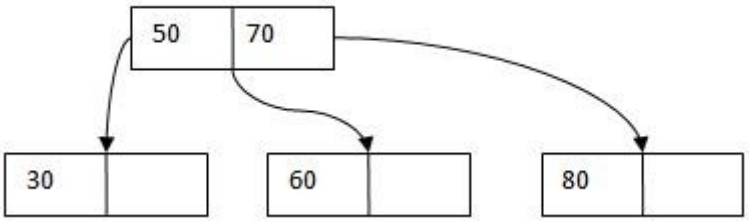


Örneğin azami anahtar sayısıs 2 olan yukarıdaki örnek ağaçta ekleme işlemi yapalım ve değer olarak 60 ekleyelim:



Yukarıdaki ekleme işlemi, ekleme algoritmamızdaki 2. durumda gerçekleşmektedir. Yani anahtarımızın ekleneceği yaprakta boş yer bulunmaktadır ve buraya yeni anahtarı ekleriz.

Şayet yukarıdaki ağaca 80 değerini ekleyecek olsaydık bu durumda da algoritmamızdaki 3. ihtimal gerçekleşmiş olacaktı.



Görüldüğü üzere 80 anahtarının ekleneceği düğüm dolmuş ve azami 2 anahtar olamsı gerekirken 3 anahtar olmuştur. Ortanca değer (median) 70 olan bu ekleme işleminden sonra ortanca değer bir üst düğüme çıkmış ve iki farklı düğüme ortanca elemanın solundaki ve sağındaki değerler yukarıdaki şekilde dağıtılmıştır.

5. B Ağacından Silme

B ağacı yukarıdaki özellikleri bölümünde anlatılan özelliklerin bozulmaması için silme işlemi sırasında aşağıdaki iki yöntemden birisini izler:

1. çözümde ağaçtan ilgili anahtar bulunup silinir ve bütün ağaç yeniden inşa edilir

2. çözümde ağaçtan ilgili anahtar bulunup silinir ve bulma işlemi sırasında geçilen ağacın kısımları yeniden inşa edilir.

Ayrıca B+ ağacı (B plus tree) ve B# ağacı (B number tree) şeklinde alt çeşitleri de bulunmaktadır.

Ayrıca B ağacının özel birer uygulaması olarak aşağıdaki ağaçlara bakabilirsiniz:

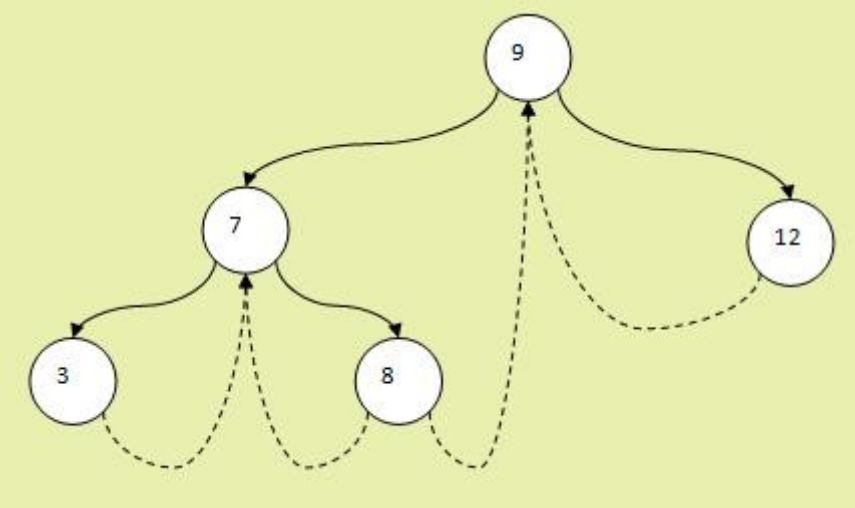
2-3 Ağacı

2-3-4 Ağacı

SORU 41: Dikişli Ağaçlar (Threaded Tree)

Dikişli ağaçlar, ikili ağaçların özel bir halidir. Bilindiği üzere ikili ağaçların son elamanı olan yapraklarda (leaf) bulunan üyeleri sol ve sağ çocuğu olarak boş (null) değer gösterirler. Dikişli ağaçlar ise bunun aksine ağaç içerisinde kimin yerine ikame edecekse bu düğümü gösterir.

Örneğin aşağıdaki ağacı ele alalım:



Yukarıdaki ikili ağaçta sonda bulunan yapraklar daha yukarıdaki silinme durumunda alternatifleri olan düğümlerii göstermiştir. Bir yaprağın çocukları yerine gösterdiği bu düğümler silinme durumunda ikame edilecek düğümlerdir. Basitçe, örneğin yukarıdaki tasvirde 7 değerine sahip düğüm silinirse yerine 8 veya 3 gelebilir. Benzer şekilde 9 numaralı düğüm silinirse yerine 8 veya 12 nmaralı düğümler gelir.

Bir düğüm silindiğinde yerine gelecek alternatif düğüm iki türlü hesaplanabilir.

Soldakilerin en büyüğü (max of left)

Sağdakilerin en küçüğü (min of right)

İşte bu iki yöntemle bir düğümün sol veya sağındaki değerler hesaplanarak kendisini göstermesi sağlanrısa bu ağaçlara dikişli ağaç (threaded tree) denilir.

Şayet bu değerlerden sadece birisi düğümü gösteriyorsa bu durumda:

sol dikişli (left threaded) : soldakilerin en büyüğü olması durumu

sağ dikişli (right threaded): sağdakileirn en küçüğü olması durumu

şeklinde özel olarak da isimlendirilirler.

Bu ağaçlar dolaşma, arama ve silme işlemlerinde kolaylık sağlamaktadırlar.

SORU 42: Bağımsız düğümler (Anti Clique, Independent Set)

Klik yapısının tersi olarak düşünülebilir. Basitçe bir grafta birbiri ile doğrudan bağlantısı olmayan düğümlerin oluşturduğu alt graftır.

Yukarıdaki tasvirde iki adet graf verilmiştir. Üstte bütün graf görülmekte altta ise bu grafın bir alt grafı görülmketedir. Dikkat edilirse sadece altta bulunan {A,E,F} düğümleri alındığında

aşağıdaki graf elde edilir ve bu grafta bulunan düğümleri birbirlerine bağlayan kenar bulunmamaktadır.

SORU 43: Klik (clique)

Graf teorisinde her iki düğümü birbirine bir kenar ile bağlanmış alt graflara verilen isimdir. Örneğin aşağıdaki grafikte bir klik kırmızı çizgiler ile işaretlenmiştir. Buna göre {A,B,C,D} alt grafı bir kliktir.

Sosyal bilimlerde de aynı kelime(klik) bir toplumun en alt birimine verilen isimdir. Bunun sebebi doğrudan bağlantısı olan ve komşuluğu bulunan bireylerden oluşması olarak yorumlanabilir.

Klik yapısının tersi için bağımsız düğümler veya anti clique veya independent set terimleri kullanılır.

SORU 44: k-düzenli graf (k-regular graph)

Bir graf üzerindeki her düğümün "k" kadar komşusu bulunması durumuna k-düzenli graf denilir. Örneğin aşağıdaki graf 2-düzenli bir graftır çünkü her düğümün derecesi 2′dir.

SORU 45: Güçlü Bağlı Graf (Strongly Connected Graph)

Bir grafta bulunan bütün düğümleri diğer bütün düğümlere bağlayan birer kenar bulunuyorsa bu grafa güçlü bağlı graf adı verilir.

SORU 46: Basit Döngü (Simple Cycle)

Bir graftaki bir döngünün başlangıç ve bitiş düğümleri olan düğümü dışındaki bütün düğümlerin, bu döngü içerisinde sadece bir kere geçmesi durumunda bu döngüye basit döngü adı verilir.

SORU 47: Bağlı graf (conected graph)

Bir graftaki bütün düğümleri diğer bütün düğümlere bağlayan bir yol bulunuyorsa bu graflara bağlı graf denilir.

SORU 48: Döngü (Cycle)

Graf teorisinde bir düğümden başlayıp aynı düğümde biten yola döngü adı verilir

graf41.jpg

Örneğin yukarıdaki grafta A düğümünden başlayarak gene bu düğümde biten {A,C,D} döngüsü tasvir edilmiştir.

SORU 49: Altgraf (Subgraph)

Bir grafikte bulunan düğüm ve kenarlardan sadece bir kısmını içeren grafa verilen isimdir. Her altgraf da bir graftır. Ayrıca grafın kendisi de altgraflarından bir tanesidir.

graf5.jpggraf51.jpg

Örneğin yukarıdaki şekilde bir graf ve bir alt grafı yanyana gösterilmiştir.

SORU 50: Yol (Path)

Bir graf üzerinde bir veya daha fazla düğümden ve kenardan geçen rotaya verilen isimdir. Örneğin aşağıdaki graf üzerinde bir yol gösterilmiştir.

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

SORU 51: Yönlü Graflar (Directed Graphs)

Bir grafın kenarlarının yön belirtmesi durumunda bu grafa yönlü graf adı verilir.

Bir kenar iki düğümü birleştirmektedir. Yönlü bir kenar ise bir düğümden diğer düğüme gidilebilen yönü göstermektedir. Bu kenarın gösterdiği yönün tersine doğru da hareket edilebilmesi durumunda bu ikinci bir kenar ile ifade edilir.

Yukarıda A ile B düğümleri arasında her iki yönde de hareket edilebildiğini gösteren iki adet kenar bulunmaktadır. Bu kenarlardan birsinin bulunmaması durumunda;

tek yönlü hareket etmek mümkün olurken tersi yönde hareket mümkün değildir. Örneğin yukarıdaki şekilde A düğümünden B düğümüne geçiş mümkün iken tersi olan B düğümünden A düğümüne hareket edilememektedir.

SORU 52: Yönsüz graflar (undirected graphs)

Bir grafta bulunan kenarların yön bildirmemesi durumunda bu grafa yönsüz graf denilir. Bu durumda iki düğüm arasında bulunan kenar, her iki yönlü de hareket edilebileceğini ifade eder.

Örneğin yukarıdaki graf yönsüzdür. Bu grafta A ile B düğümleri arasında bir a kenarı bulunmaktadır. Bu kenar yönsüz olduğu için hem A'dan B düğümüne hem de B'den A düğümüne hareket etmenin mümkün olduğunu gösterir.

SORU 53: Graf (Şekil, Graph)

Bilgisayar dünyasında bulunan ve gerçek hayatta çeşitli sebeplerle karşılaşılan yapıları temsil amacıyla kullanılan şekillerdir.

Örneğin bir bilgisayar ağını, karakenarları haritasını veya bir karar ağacını graflar kullanarak temsil etmek mümkündür.

Bilgisayar bilimleri çeşitli uygulamalarda karşılaşılan bu yapıları ifade etmek için çeşitli matematiksel ve görsel yöntemlerden faydalanır.

Buna göre bir grafta bulunan varlıklar düğümler ile ifade edilmekte, bu varlıklar arasındaki ilişkiler ise graftaki kenarlar ile ifade edilmektedir.

Grafları kenarların yönlü olup olmamasına göre, yönlü graflar ve yönsüz graflar olarak ikiye ayırmak mümkündür. Ayrıca kenarların değer almasına göre değerli graflar veya değersiz graflar isimleri verilebilir.

Örneğin yukarıdaki grafta 4 düğümden ve 4 kenardan oluşan bir graf gösterilmektedir. Bu grafı G= ({A,B,C,D} , {(A,B),(A,C),(C,D),(A,D)}) şeklinde ifade etmek mümkündür. Dolayısıyla graflar G= (V,E) şeklinde yanı düğümler ve kenarlar şeklinde yazılmaktadır.

SORU 54: Kenar (Edge)

Bir graf üzerindeki her çizgiye kenar adı verilir. kenarlar düğümleri birleştirdikleri için bu ismi almışlardır. Graf teorisinde bir kenaru ifade etmek için birleştirdiği düğümlerin isimleri yazılır. Örneğin aşağıdaki şekildeki “a” kenarunu ifade etmek için (A,B) gösterimi kullanılır.

SORU 55: Düğüm (Node)

Bir graf üzerindeki her noktaya düğüm adı verilir. Düğümler, kenarlar kendi üzerlerinde birleştiği için bu ismi almışlardır.

Graf teorisine göre bir düğümün derecesi o düğümde bulunan kenar sayısıdır. Örnepin aşağıdaki grafta A düğümünün derecesi 3′tür.