CENG405
Bilgisayar Bilimlerinde Güncel Konular - I

Ders Notları

Prof. Dr. Serdar İplikçi

İÇİNDEKİLER

1) VERİ MADENCİLİĞİNE (DATA MINING) GİRİŞ

Günümüzdeki Internet, Endüstri 4.0, Dijital Dönüşüm ve Nesnelerin Interneti (IoT) kavramları ile birlikte her geçen gün veri miktarı artmaktadır. 2020 yılı itibarıyla yapılan bir araştırmaya göre her gün 2.5×10^9 GB (2.5 quintillion GB) veri üretilmektedir ve bu rakam her geçen gün artmaktadır. Şu ana kadar üretilmiş veri miktarının 44 zettabyte olduğu tahmin edilmektedir. Birimler ile ilgili tablo aşağıda görülmektedir:

Kısaltma	Birim	Değer	Boyut (byte cinsinden)
b	bit	0 veya 1	1/8 byte
B	byte	8 bit	1 byte
KB	Kilobyte	1000 bytes	1.000 bytes
MB	Megabyte	1000^2 bytes	1.000.000 bytes
GB	Gigabyte	1000^3 bytes	1.000.000.000 bytes
TB	Terabyte	1000^4 bytes	1.000.000.000.000 bytes
PB	Petabyte	1000^5 bytes	1.000.000.000.000.000 bytes
EB	Exabyte	1000^6 bytes	1.000.000.000.000.000.000 bytes
ZB	Zettabyte	1000^7 bytes	1.000.000.000.000.000.000.000 bytes
YB	Yottabyte	1000^8 bytes	1.000.000.000.000.000.000.000.000 bytes

Bu verilerin büyük çoğunluğu, sosyal medya paylaşımları, mobil cihazlardaki uygulama kullanımları, web sayfalarında bırakılan loglar, nesnelerin interneti sayesinde oluşan sensör verileri vb. gibi insanların ve nesnelerin dijital ayak-izlerinden üretilmektedir. Günümüzün rekabetçi ortamında, verilerden elde edilebilecek bilginin stratejik önemi tartışılmazdır. Bunun için son yıllarda Veri Madenciliği ve Büyük Veri gibi teknolojiler geliştirilmiştir.

Literatürde Veri Madenciliği ile ilgili pek çok tanım bulunmaktadır: Örnek olarak;

- “Veri madenciliği; büyük miktarda eldeki yapısız veriden, geçmişteki bilgileri analiz ederek -kümeleme, veri özetleme, sapma tespiti gibi teknik yaklaşımlarla- anlamlı ve kullanışlı bilgiye ulaşarak, gelecekle ilgili tahmin yapmaya yönelik çalışmalardır.”
- “Veri madenciliği; ham, kullanışsız verileri faydalı bilgilere dönüştürüp, trend ve davranış analizine dayalı otomatik kalıp tahminleri yaparak kullanım alanına göre farklı şekillerde kullanıcısına avantaj sağlayan işlemler bütünüdür.”
- “Veri madenciliği; büyük ve kullanışsız bir veri tabanından, makine öğrenimi ve istatistik bilimi kullanımıyla, istenilen veriyi seçmeye ve oluşturulan şablonlarla gelecek kullanımlar için işlemeye yönelik çalışmalar bütünüdür.”
- “Veri madenciliği, çevremizde olanları daha iyi anlayabilmek veya ileriye dönük tahminlerde bulunabilmek için çok geniş bir veri ağı içerisindeki verileri gerekli işlemlerden geçirerek faydalı olması muhtemel bilginin diğerlerinden ayıklanarak ortaya çıkarılması işlemidir.”
- “Veri madenciliği; çok fazla değersiz verinin bir takım teknikler ve aşamalarla bu veriler arasındaki ilişkileri daha çok netleştirerek şimdi ve geleceğimizle ilgili daha önemli bilgileri keşfetmemizi sağlar.”
- “Veri madenciliği; büyük veri birikimleri arasından gelecekle ilgili öngörülerde bulunabilmemizi sağlayacak verileri programlama ve istatik bilimini kullanarak ayırmasıdır.”

Diğer taraftan, eğer üzerinde çalışacak veriler aşağıdaki beş özelliğin hepsine sahipse, o zaman veri madenciliği problemi “Büyük Veri” problemine dönüşmektedir:

- Volume (Hacim): Bir verinin 'büyük veri' olup olmamasının en önemli şartı yüksek boyutlarda olmasıdır.Verinin boyutu verinin değerini belirler.
- Variety (Çeşitlilik): Verilerin belirli bir biçimde olmasına gerek yoktur. Veriler, resim (image), metin (text), ses (audio), video, log dosyaları gibi bir çok veri formatında olabilir.
- Velocity (Hız): İşlenecek veri miktarı sabit olmayıp her geçen birim zamanda veri miktarının artmakta olması, bir bakıma akan veri (streaming data) olmasıdır.
- Value (Değer): En önemli bileşenlerden birisi de değer katmanıdır. Analiz edilen verilerin kullanıcı için artı değer sağlıyor olması gereklidir
- Verification (Doğruluk): Bu kadar hızlı ve büyük olan verilerin akışı sırasında, gelen verilerin güvenli olup olmadığını kontrol etmek gerekir. Aksi halde, kirli ve bozulmuş verinin depolanması ve daha sonra analiz edilmesi ilave zaman kaybı ve hatalı sonuçlara yol açabilir.

Özetlersek, veri madenciliğinin amacı, geçmişi anlayıp geleceği tahmin etmeye çalışmaktır. Bir büyüklüğün gelecekteki değerini tahmin etmek için geliştirilen yapay sinir ağları, destek vektör makineleri gibi yöntemler "açıklayıcı (explanatory)" olmaktan çok giriş-çıkış verisini doğru bir şekilde temsil etmeye dayalı modeler olduğu için genellikle kara-kutu (black box) modeler gibi davranırlar. Oysa ki veri madenciliği yöntemleri, veriler içerisindeki örüntüleri bularak insanlar tarafından kolayca anlaşılabilecek ve yorumlanabilecek sonuçlar üretmeyi hedeflemektedir. Veri içerisindeki örüntüleri bulmak için literatürde önerilmiş olan yöntemler, bulunacak örüntülerin tipine göre çeşitlilik kazanmaktadır. Yaygın olarak kullanılan bazı örüntü tipleri olarak, kümeler (clusters), öğesetleri (itemsets), eğilimler (trends) ve aykırılıklar (outliers) verilebilir.

Literatürde mevcut Veri Madenciliği algoritmaları şu şekilde gruplandırılabilir: ilişkisel kural madenciliği (association rule mining), öğeseti madenciliği (itemset mining), sıralı örüntü madenciliği (sequential pattern mining), sıralı kural madenciliği (sequential rule mining), sekans tahmini (sequence prediction,), periyodik örüntü madenciliği (periodic pattern mining), episod madenciliği (episode mining), yüksek-faydalı örüntü madenciliği (high-utility pattern mining), zaman serisi madenciliği (time-series mining), kümeleme ve sınıflandırma (clustering and classification.)

Bu derste daha çok, bir veritabanındaki sık öğesetlerinin bulunması ve bunlar arasındaki ilişkilerin ortaya çıkarılması için geliştirilmiş algoritmalar üzerinde durulacaktır.

2) VERİ MADENCİLİĞİ YAKLAŞIMLARI ve UYGULAMALARI

Veri Madenciliği Uygulamaları şu şekildedir:

• Veri madenciliğinin; Eğitim, Üretim mühendisliği, Müşteri ilişkileri yönetimi, Dolandırıcılık Tespiti, İzinsiz giriş tespiti, Yalan Algılama, Müşterileri Sınıflandırma, Suç Soruşturması, Araştırma analizi ve bunlara benzer başka bir çok kullanım alanı vardır. Aynı zamanda en yaygın kullanım alanlarından birisi olan müşteri ilişkilerinin yönetimine bir örnek olarak; işletmeler müşterileri hakkında daha fazla bilgi edinebilir ve çeşitli işletme işlevleriyle ilgili daha etkili stratejiler geliştirebilir ve dolayısıyla kaynakları daha optimal ve anlayışlı bir şekilde kullanabilir.

• Bir firmanın yapacağı kampanyanın daha etkili olabilmesi için yaptığı satışların hangi kategori de daha fazla olduğunu belirleyip geleceğe dönük tahmin yapılabilir.

• Veri madenciği mesela bir pazar araştırmasında kullanılabilir ve müşteriler arası benzerlikler veya sepet analizi gibi yöntemler kullanılabilir. Diğer kullanım alanları ise pazarlama, bankacılık ve finans sektörü, e-ticarette, sigortacılıkta, spor bilimlerinde, sağlık ve ilaç sektöründe kullanılır.

• Veri madenciliği; finans, eğitim, reklamcılık, sağlık, sosyal hizmetler, mühendislik gibi bir çok temel sektörde; müşteri profili oluşturmak, öğrenim alışkanlıklarını incelemek, hastalık teşhis ve davranışlarını takip etmek, bilimsel analizler yapmak gibi önemli kullanım olanakları sağlar.

• Veri madenciliği;web üzerinde filtrelemeler, elektronik alışverişte müşteri alışkanlıklarına göre öneriler, ekonomideki eğitim ve düzensizlik tespitlerinde ayrıca DNA sıraları içerisinde gen tespiti gibi uygulama alanlarında çalışmalar yapar.

• Eğitim, tıp, biyoloji, güvenlik, üretim, ticari ve finansal alanlar başta olmak üzere çok çeşitli alanlarda pazar araştırması, risk analizi ve kaynakların en iyi şekilde kullanımı gibi konularda veri madenciliğine başvurulabilinir.

3) SIK ÖĞESETİ MADENCİLİĞİ (FREQUENT ITEMSET MINING - FIM) ALGORİTMALARI

3.1) Sık Öğeseti Madenciliğine Giriş

Veri içindeki örüntülerin bulunması fikri ilk olarak 1993'te Agrawal ve ark. tarafından ortaya atılmıştır [1]. Buna ilk olarak büyük örüntü madenciliği denilmiş olsa da artık günümüzde buna Sık Örüntü Madenciliği (Frequent Itemset Mining - FIM) denmektedir. FIM ilk olarak market sepeti verilerinin analizinde kullanıldığı için, FIM kavramını tanımlamak için de market sepeti uygulamasından yararlanarak şu şekilde tanımlama yapılabilir: müşterilerin yaptıkları alış-veriş ya da işlemlerin (transactions) olduğu bir veritabanı (database) verildiğini düşünelim. Burada FIM birlikte satın alınan öğe veya öğe-kümelerinden en sık görülenleri bulmaya çalışır. Örnek olarak, bir FIM algortimasının sonucunda, çok sayıda müşteri tarafından kurabiye ile baharatın bir arada satın alındığı gibi bir örüntü ortaya çıkabilir. Bu öğeler arasındaki ilişkilerin ortaya çıkarılması, müşteri davranışlarının analiz edilmesinde, pazarlama konusunda (bir arada çokça satılan ürünlerin rafta yanyana konması veya kampanya yapılması gibi) stratejik kararlar alınmasında oldukça faydalıdır.

FIM her ne kadar ilk olarak müşteri davranışlarının analiz edilmesi için önerilmiş olsa da, artık günümüzde pek çok alana uygulanabilecek bir veri madenciliği işi olarak görülmektedir. Zira, müşteri hareketlerinin olduğu veritabanı daha da genelleştirilerek FIM daha farklı alanlarda kullanılabilmektedir. Artık bu daha genel veritabanında, müşteri hareketleri yerine belli özelliklere (attribute) sahip nesneleri (objects) tanımlayan örnekler (instances) bulunacaktır. Böylece, FIM, veritabanında bir arada sık görülen özellikleri bulma işi olarak daha genel bir şekilde tanımlanabilir. Pek çok veri tipi, işlem veritabanı (transaction database) şeklinde temsil edilebildiği için, FIM, bioinformatics, resim sınıflandırma, network trafik analizi, müşteri yorumlarının analizi, aktivite izleme, zararlı yazılımların tesbiti ve e-learning gibi çok çeştli alanlarda uygulanabilmektedir. Ayrıca, FIM, rare patterns, correlated patterns, patterns in sequences and graphs, ve that high-profit patterns gibi özel tip örüntülerin bulunmasında da kullanılabilir. FIM, sürekli yeni algoritmaların geliştirildiği çok aktif bir araştırma alanıdır.

3.1.1) Problemin Tanımı

FIM Algoritmalarının amacı, verilen bir işlem veritabanındaki önemli/ilginç öğe ve öğesetlerini ya da başka bir deyişle örüntüleri bulmaktır. İşlem veritabanına örnek olarak bir markette belli bir dönemde müşteriler tarafından yapılan alış-verişlere ilişkin alış-veriş fişlerinden oluşan bir veritabanı verilebilir. Örneğin bu markette belli bir dönemde müşteriler tarafından 5 adet alış-veriş yapılmış olsun ve her bir alış-verişte satın alınan ürünler ve miktarları aşağıdaki tablodaki gibi verilsin:

Fiş No	Satın Alınan Ürünler
#1	1 kg `Armut`, 2 paket `Cips`, 1 kutu `Diş Macunu`
#2	10 adet `Bardak`, 3 paket `Cips`, 2 litre `Elma Suyu`
#3	2 kg `Armut`, 15 adet `Bardak`, 3 paket `Cips`, 1 litre `Elma Suyu`
#4	20 adet `Bardak`, 5 litre `Elma Suyu`
#5	3 kg `Armut`, 5 adet `Bardak`, 1 paket `Cips`, 1 litre `Elma Suyu`

Bu tabloda her bir alış-veriş fişi içinde yer alan ürünler ve miktarları görülmektedir. Şimdi, verilen bu alış-veriş veritabanından, işlem (transaction) veritabanını elde edelim. Alış-veriş veritabanındaki her bir fiş, işlem veritabanında bir işleme transaction) karşı düşmektedir. Ürünlerin isimlerini de ürünün sadece ilk harfini kullanarak kısaltalım, yani, armut A, bardak B, cips C, diş macunu D ve elma suyu da E harfiyle sembolize edilsin. Ayrıca, bir fişteki ürünün miktarını (şimdilik) göz ardı edelim, yani sadece o ürün o fişte var mı yok mu, sadece buna bakalım. Böylece, işlem veritabanı aşağıdaki gibi elde edilir:

İşlem ID (TID)	İşlem (Transaction)
T_1	A, C, D
T_2	B, C, E
T_3	A, B, C, E
T_4	B, E
T_5	A, B, C, E

Görüldüğü gibi, her bir fiş, bir işlem olarak ele alınmış, her bir fişteki ürünün miktarına bakılmadan o fişte yer alan ürünler işlemde kendi sembolleri ile yer almıştır.

Bu örnekte her bir ürün bir öğeyi temsil etmektedir. Birden fazla farklı öğenin bir araya gelmesiyle öğesetleri oluşmaktadır. Örneğin, {A, D, E} birlikte bir öğesetidir. Dolayısıyla, bir FIM algoritmasıyla sadece tek bir öğe değil aynı zamanda bu öğelerin olası tüm kombinasyonlarından oluşan öğesetlerinin en sık görülenleri bulunacaktır.

Önce bazı gerekli tanımlarla başlayalım. $I = \{x_1 \quad x_2 \quad \cdots \quad x_n\}$ şeklinde öğelerden oluşan bir öğe kümesi ele alalım. $X = \{x_1 \quad x_2 \quad \cdots \quad x_m\}$ öğeseti, bu I öğe kümesinin bir alt kümesi olsun, yani $X \subseteq I$, X öğeseti aynı zamanda $X = x_1 x_2 \dots x_m$ şeklinde de gösterilebilir. Bir işlem (transaction) $T = (tid, X)$ şeklinde verilen bir ikilidir (tuple), burada tid işlem kimliği (transaction id) ve X de bir işlem ya da öğesetidir. Bu T işlemi Y öğesetini ancak ve ancak Y$\subseteq X$ olması durumunda "T işlemi Y öğesetini içerir" denilebilir. Bir TDB işlem veritabanı (transaction database) işlemlerden oluşan bir veritabanıdır. T işlem veritabanında yer alan X gibi bir öğesetini içeren işlem sayısına X öğesetinin *mutlak destek* (*absolute support*) değeri denir ve $AbsSupp(X)$ ile gösterilir.

Verilen bir T işlem veritabanı ve minimum destek değeri $MinSupp$ değeri için, $AbsSupp(X) \geq MinSupp$ şartını sağlayan bir öğesetine *sık öğeseti* (*frequent itemset*) denir. Sık öğeseti yerine *sık örüntü* (*frequent pattern*) ifadesi de kullanılmaktadır. *Sık öğeseti* (*örüntü*) *madenciliği*, ya da *frequent itemset* (*pattern*) *mining* problemi, verilen bir T işlem veritabanı ve minimum destek değeri $MinSupp$ değeri için tüm sık öğesetlerinin bulunmasıdır.

FIM algoritmalarının amacı, verilen bir veritabanındaki önemli/ilginç örüntüleri ortaya çıkarmak olduğuna göre, bu kavramın tanımlanması gerekir. Genel olarak, bir örüntünün önemliliği ya da ilginçliği için çeşitli ölçütler ortaya konsa da FIM algoritmalarında önemlilik/ilginçlik ölçütü olarak *destek* (*support*) kavramı kullanılır. Verilen bir D gibi bir veritabanında, X gibi bir öğesetinin *mutlak-desteği* (absolute support) $AbsSupp(X)$ ile gösterilir ve X öğesetini içeren işlemlerin (transactions) sayısı olarak tanımlanır yani,

$$AbsSupp(X) = |\{T \mid X \subseteq T \ \wedge \ T \in D\}| = X \text{ öğesini içeren işlem sayısı}.$$

Benzer şekilde, verilen bir D gibi bir veritabanında, X gibi bir öğesetinin bağıl-desteği (relative support) $RelSupp(X)$ ile gösterilir ve N işlem sayısı (number of transactions) olmak üzere, X öğesetini içeren işlemlerin (transactions) sayısının toplam işlem sayısına (N) oranı olarak tanımlanır yani,

$$RelSupp(X) = \frac{AbsSupp(X)}{N} = \frac{X \text{ öğesini içeren işlem sayısı}}{\text{toplam işlem sayısı}}.$$

TID	İşlem (Transaction)
T_1	A, C, D
T_2	B, C, E
T_3	A, B, C, E
T_4	B, E
T_5	A, B, C, E

Örnek olarak, yukarıdaki Tablo'da verilen veritabanı için {A, B} öğesetinin mutlak-desteği 2, bağıl-desteği 0.4'tür, çünkü {A, B} öğeseti sadece T_3ve T_5 işlemlerinde yer almaktadır, yani,

AbsSupp({A, B})=2 *RelSupp*({A, B})=2/5

Şimdi de *sık* (*frequent*) öğeseti kavramını ele alalım. *minsup* gibi verilen bir minimum destek (eşik) değeri için, X gibi bir öğesetinin destek değeri bu eşik değerine eşit veya daha fazlaysa, o zaman bu X gibi öğesetine *sık öğeseti* (*frequent itemset*) denir. Yani, eğer $RelSupp(X) \geq minsupp$ (veya $AbsSupp(X) \geq minsupp$) ise, o zaman X bir sık öğesetidir (frequent itemset).

FIM algoritmalarının amacının, verilen bir veritabanındaki tüm sık öğe-kümelerini bulmak olduğu daha önce belirtilmişti. Örnek olarak, Tablo 1'de verilen işlem veritabanını ele alalım. Bu işlem veritabanı için $minsupp = 3$ olmak üzere, tüm sık öğe-kümeleri, mutlak destek değerleri karşılarında belirtildiği gibi, şu şekildedir:

	Öğeseti	**AbsSupp**		**Itemset**	**Öğeseti**
#1	{A}	3	#6	{B, C}	3
#2	{B}	4	#7	{B, E}	4
#3	{C}	4	#8	{C, E}	3
#4	{E}	4	#9	{B, C, E}	3
#5	{A, C}	3			

Dolayısıyla, FIM bir sayma (enumeration) problemidir. Amaç, minimum support şartını sağlayan tüm öğe-kümelerini ortaya çıkarmaktır. Böylece, oldukça zor olan FIM probleminin her zaman tek bir çözüm kümesi vardır. FIM problemini çözmek için izlenecek en basit (naive) yöntem, önce olası tüm öğe-kümelerini bulup ardından bunların destek (support) değerlerini bularak *minsup* eşiğini geçenleri *sık öğeseti* (*frequent itemset*) döndürmek şeklinde olan *Brute-Force* yaklaşımıdır. Bilgisayar Bilimleri'nde brute-force arama yaklaşımı (veya exhaustive search yaklaşımı), oluştur ve test et (generate-and-test) olarak da adlandırılır, olası tüm aday çözümlerin teker teker bulunup denenmesi esasına dayanmaktadır. Ancak, böyle basit bir yaklaşım şu sebepten dolayı kullanışsızdır: diyelim ki veritabanında m farklı öğe olsun, o zaman bu veritabanında $2^m - 1$ farklı öğeseti olacaktır. Örneğin, A, B, C, D ve E öğelerinin bulunduğu bir veritabanının *öğeseti uzayı*'nda (*itemset space*) $2^5 - 1 = 31$ farklı öğeseti olacaktır:

Eğer bir veritabanında 100 farklı öğe olursa, o zaman $2^{100} - 1$ farklı öğeseti olacaktır ve brute-force yaklaşımıyla sık öğe-kümelerinin bulması neredeyse imkansız hale gelecektir. Bu yaklaşım, çok küçük veritabanlarında bile kullanışsızdır. Örneğin, $minsupp = 1$ olmak üzere, 100 farklı öğeden oluşan tek bir işlemin (transaction) olduğu bir veritabanı için bile $2^{100} - 1$ farklı öğeseti oluşturmaya çalışacaktır. Dolayısıyla, genel olarak, veritabanındaki öğe sayısı, veritabanındaki işlem sayısından daha önemli hale gelmektedir. Peki, arama uzayındaki öğeseti sayısını hangi faktörler etkilemektedir? Öğeseti sayısı, hem *minsupp* değerine, hem de veritabanındaki öğelerin birbirine ne kadar benzediğine bağlıdır

Sık öğe-kümelerini bulmak için literatürde etkili yöntemler geliştirilmiştir. Bu algoritmalar, öğeseti uzayındaki olası tüm öğe-kümelerini araştırmaktan kaçınarak sık öğe-kümelerini olabildiğince etkili bir şekilde bulmaya çalışırlar. Bunlardan bazıları, Apriori, ECLAT, H-mine ve FPtree algoritmalarıdır. Tüm bu algoritmalar, aynı veritabanı ve *minsupp* değerine karşı aynı sık öğeseti ve destek değeri çıkışını üretirler. Bu algoritmalar arasındaki farklar algoritmaların stratejileri ve kullandıkları veri yapılarıdır. Daha özele inilecek olursa, FIM algoritmaları şu şekilde ayrılırlar:

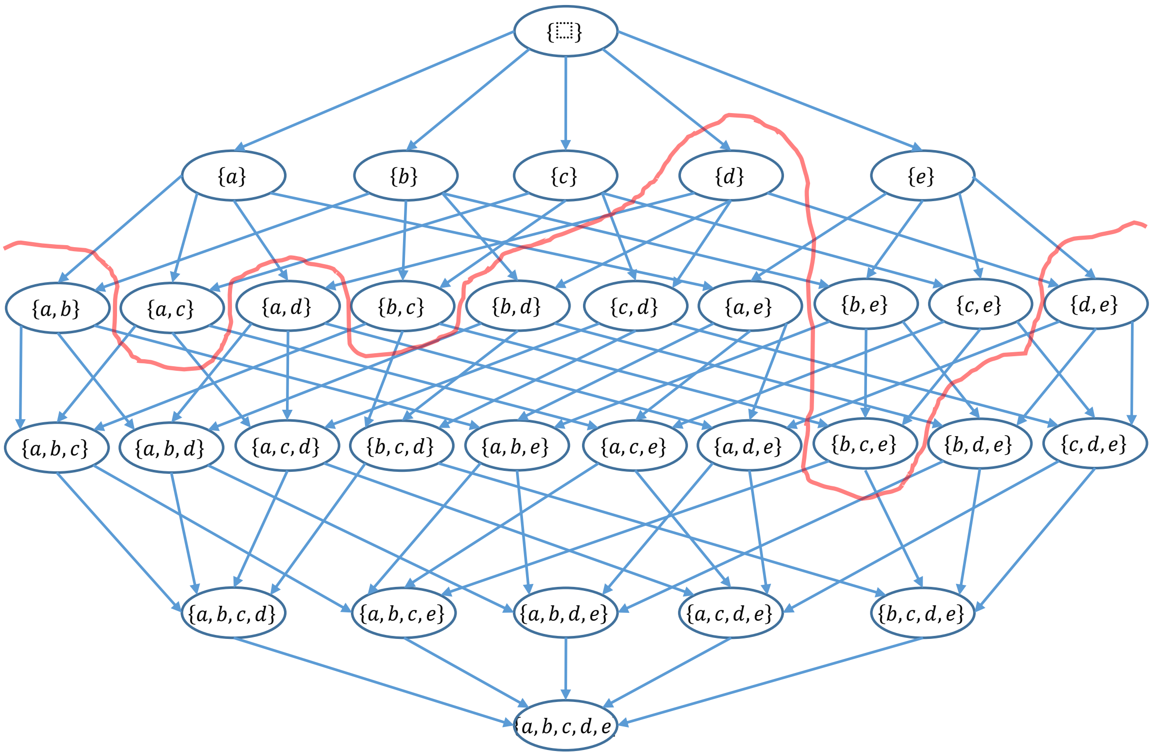
(1) Hangi tip arama yöntemini kullanmaktadır. İki tip arama yöntemi vardır. *depth-first search* ve *breadth-first search*.
(2) Kullanılan veritabanının tipi. Yatay (horizontal) veya dikey (vertical) olabilir.
(3) Arama uzayında aranacak bir sonraki öğesetinin oluşturulma biçimi.
(4) Bir öğesetinin sık olup olmadığını anlamak için destek (support) değerinin nasıl hesaplandığı.

3.1.2 Genişlik-Öncelikli Arama (Breadth-First Search)

Varsayalım ki veritabanında m farklı öğe bulunsun. Bir breadth-first search tipindeki algoritma (aynı zamanda *level-wise* algoritma da denir) öğeseti uzayındaki aramayı şu şekilde yapar: Önce 1 uzunluklu öğe-kümelerini yani 1-öğe-kümelerini (1-itemsets) bulur. Ardından, 2 uzunluklu öğe-kümelerini yani 2-öğe-kümelerini (2-itemsets) bulur ve bu şekilde m-öğe-kümelerinine (m-itemsets) kadar devam eder. Örnek olarak, aşağıdaki Tablo ile verilen veritabanını ele alalım.

TID	İşlem (Transaction)
T_1	$\{a, c, d\}$
T_2	$\{b, c, e\}$
T_3	$\{a, b, c, e\}$
T_4	$\{b, e\}$
T_5	$\{a, b, c, e\}$

Bu veritabanına ilişkin olası tüm öğe-kümelerinin arama uzayı aşağıdaki şekilde görülmektedir. Şekilde bu arama uzayı bir *Hasse diagramı* şeklinde verilmiştir. Bir *Hasse diagramı*, ancak ve ancak $X \subseteq Y$ ve $|X| + 1 = |Y|$ ise, X öğesetinden Y öğesetine bir ok çizer.

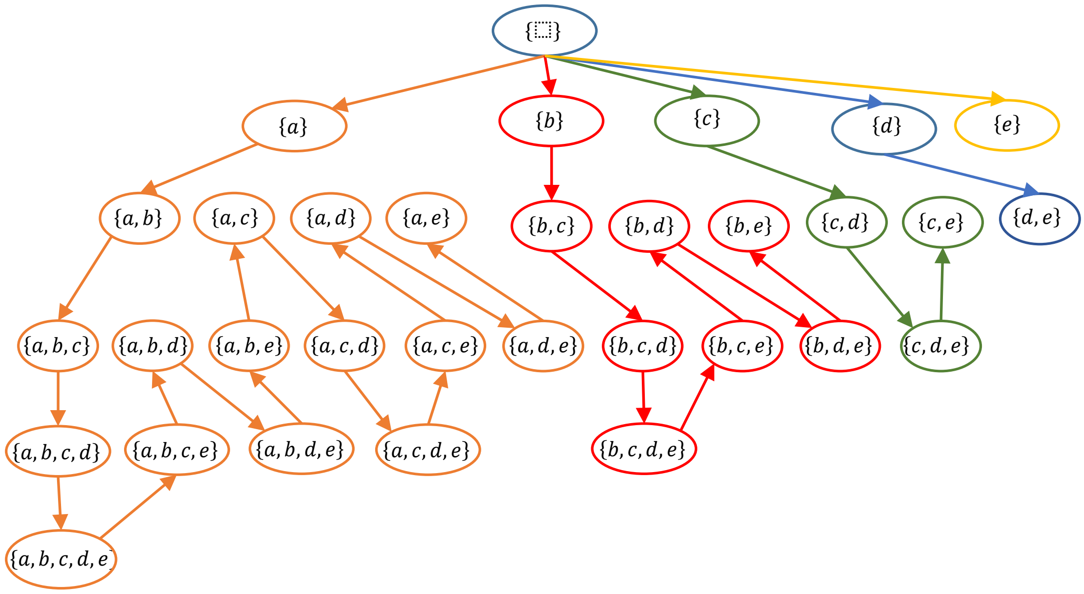


Şekil 3.1 $I = \{a, b, c, d, e\}$ kümesi için breadth-first search arama uzayı (*Hasse Diyagramı*)

3.1.3 Derinlik-Öncelikli Arama (Depth-First Search)

Bilindiği gibi, bir breadth-first search algoritması ilk olarak 1-öğe-kümeleri olan a, b, c, d and e öğe-kümelerini bulur, ardından $\{a, b\}$, $\{a, c\}$, $\{a, d\}$ gibi 2-öğe-kümelerini bulur, sonra 3-öğe-kümelerini bulur ve bu işlem tüm öğeleri içeren ve son öğeseti olan $\{a, b, c, d, e\}$ öğesetine kadar devam eder. Diğer taraftan, tipik bir depth-first search algoritması her bir 1-öğesetinden başlar ve sonra tekrarlı bir şekilde (recursively) o anki öğesetine öğeler ekleyerek daha büyük öğe-kümeleri oluşturur. Örnek olarak, Tablo 6.2 ile verilen örnekteki veritabanı ele alınırsa, tipik bir depth-first search algoritması, Şekil 6.2'de de görüldüğü gibi, öğe-kümelerini şu sıra ile oluşturur:

$\{a\}$, $\{a,b\}$, $\{a,b,c\}$, $\{a,b,c,d\}$, $\{a,b,c,d,e\}$, $\{a,b,c,e\}$, $\{a,b,d\}$, $\{a,b,d,e\}$, $\{a,b,e\}$, $\{a,c\}$, $\{a,c,d\}$, $\{a,c,d,e\}$, $\{a,c,e\}$, $\{a,d\}$, $\{a,d,e\}$, $\{a,e\}$, $\{b\}$, $\{b,c\}$, $\{b,c,d\}$, $\{b,c,d,e\}$, $\{b,c,e\}$, $\{b,d\}$, $\{b,d,e\}$, $\{b,e\}$, $\{c\}$, $\{c,d\}$, $\{c,d,e\}$, $\{c,e\}$, $\{d\}$, $\{d,e\}$, $\{e\}$.



Şekil. $\{a,b,c,d,e\}$ kümesinin depth-first search arama uzayı

3.1.4 Arama Uzayı Daraltma (Search Space Pruning)

Etkili bir FIM algoritması geliştirmek için, algoritmanın, tüm arama uzayını taramasından kaçınması oldukça önemlidir, çünkü önceden de belirtildiği gibi $2^m - 1$ farklı öğeseti barındıran arama uzayı çok büyük olabilir. Arama uzayını daraltmak ya da küçültmek için, bazı *arama uzayı daraltma teknikleri* (*search space pruning techniques*) kullanılmaktadır. FIM probleminde, arama uzayını daraltmak için en kritik tesbit şudur:

Destek (support) monotone bir ölçüttür, yani, X ve Y gibi iki öğeseti için, eğer $X \subset Y$ ise, o zaman $AbsSupp(X) \geq AbsSupp(Y)$ olacaktır. Bunun anlamı şudur: eğer bir öğeseti sık değilse (infrequent), o zaman onun tüm üst-öğe-kümeleri (supersets) de sık olmayacaktır ve artık bu üst-öğe-kümelerinin araştırılmasına gerek yoktur. Örnek olarak, $minsupp = 3$ olsun, $\{a,b\}$ öğesetinin desteği 2 olduğu için sık bir öğeseti değildir. Dolayısıyla, onun üst-öğesetleri olan $\{a,b\}$, $\{a,b,c\}$, $\{a,b,d\}$, $\{a,b,e\}$,$\{a,b,c,d\}$, $\{a,b,c,e\}$, $\{a,b,d,e\}$ ve $\{a,b,c,d,e\}$ öğesetleri de sık olmayacaktır ve bunları araştırılmasına gerek yoktur. Bu özelliğe, *downward-closure property*, *anti-monotonicity property* ve *Apriori property* gibi isimler verilmektedir. *Downward-closure property* arama uzayını önemli ölçüde daraltmaktadır. Bunu, Tablo 2 ile verilen veritabanı örneği ile görmek mümkündür. Normalde 31 öğesetinden oluşan arama uzayı, *downward-closure property* dikkate alındığında, yukarıdaki şekilden de görüleceği gibi, sadece 9 öğesetine indirgenmektedir.

3.2 Apriori Algoritması [1]

Yatay (horizontal) veritabanı ve breadth-first search yaklaşımı kullanan Apriori algoritması, literatürdeki ilk FIM algoritmasıdır. Giriş olarak işlem veritabanı (transaction database) ve *MinSupp* eşik değerini alır ve buna göre sık öğesetlerini ve bunların destek değerlerini çıkış olarak döndürür. Apriori algoritması aşağıdaki tabloda verildiği gibi, yatay veritabanı da (*horizontal database*) adı verilen standart veritabanı gösterilimini kullanır.

TID	İşlem (Transaction)
T_1	$\{a, c, d\}$
T_2	$\{b, c, e\}$
T_3	$\{a, b, c, e\}$
T_4	$\{b, e\}$
T_5	$\{a, b, c, e\}$

Apriori algoritmasının sözde-kodu (pseudocode) aşağıda verilmiştir. Apriori algoritması ilk olarak, satır 1'de görüldüğü gibi, her bir öğenin desteğini hesaplamak için veritabanını tarar. Ardından, buradan gelen sonuçlara göre, sık 1-öğesetlerini (*frequent 1-itemsets*) bulur, bu öğesetleri F_1 ile gösterilmektedir. Ardından Apriori algoritması, line 4-10'da görüldüğü gibi, daha büyük öğesetleri oluşturmak için tekrarlı bir şekilde breadth-first search araması yapar. Bu arama esnasında, F_{k-1} ile gösterilen $k-1$-uzunluklu sık öğesetlerini kullanarak, C_k ile gösterilen k-uzunluklu potansiyel sık öğeseti *aday* kümesini oluşturur. F_{k-1} kümesinden C_k kümesini oluşturmak için, line 5'te görüldüğü gibi, F_{k-1} kümesindeki, bir öğe hariç diğer tüm öğeleri aynı olan $k-1$ uzunluklu öğeseti çiftleri birleştirilerek C_k kümesi oluşturturulur. Bunun için iki öğesetinin birleştirilmesi kavramına yakından bakalım: $\alpha = \{a_1 a_2 \cdots a_{n-1} a_n\}$ ve $\beta = \{b_1 b_2 \cdots b_{n-1} b_n\}$ gibi aynı uzunluklu iki öğeseti alalım. α öğesetinin ilk öğesinin atılmasından arta kalan öğeseti ($\{a_2 \cdots a_{n-1} a_n\}$) ile β öğesetinin son öğesinin atılmasından arta kalan öğeseti ($\{b_1 b_2 \cdots b_{n-1}\}$) aynı ise o zaman iki öğesetinin birleştirilmesinden $\{a_1 a_2 \cdots a_{n-1} a_n b_n\}$ öğeseti elde edilir. Örnek olarak, F_1 kümesi içerisinde yer alan $\{a\}$, $\{b\}$, $\{c\}$, ve $\{e\}$ öğesetleri birleştirilerek 2-uzunluklu aday öğesetleri barındıran C_2 kümesi $\{a, b\}$, $\{a, c\}$, $\{a, e\}$, $\{b, c\}$, $\{b, e\}$ ve $\{c, e\}$ şeklinde elde edilir. Devam edersek, C_k kümesi oluşturulduktan sonra, Apriori algoritması, C_k kümesinin $(k-1)$ uzunluklu her bir alt kümesinin sık olup olmadığına bakar. Eğer, C_k kümesindeki k-uzunluklu bir X öğesetinin $(k-1)$ uzunluklu alt-kümelerinden herhangi biri sık değilse, o zaman, *downward-closure property* özelliğine göre, X öğeseti de sık değildir ve C_k kümesinden çıkarılır. Bu işlem, line 7'de görüldüğü gibi, C_k kümesindeki tüm aday öğesetleri için yapılır. Line 8'de görüldüğü gibi, *minsup* eşiğini geçen her bir aday öğeseti F_k kümesine alınır. Artık yeni aday öğeseti bulunmayana kadar bu işleme devam edilir. Son olarak, tüm sık öğesetleri ve destek değerleri çıkış olarak döndürülür.

Algoritma. Apriori Algoritması
Girdiler: T: Yatay işlem veritabanı, $MinSupp$: eşik değeri
Çıktılar: F: Tüm sık öğesetleri ve destek değerleri kümesi
Veritabanını tarayarak tüm tek öğeleri bul.
$F_1 = \{i \mid i \in I \wedge sup(\{i\}) \geq minsupp\}$
$k = 2$
while $F_k \neq \emptyset$
$C_k =$ CandiateGeneration(F_{k-1})
C_k içerisinde olup da, F_{k-1} içerisinde yer almayan $(k-1)$-uzunluklu öğeseti içeren adayları çıkar
C_k içerisindeki tüm adayların destek değerlerini bul
$F_k = \{X \mid X \in C_k \wedge sup(X) \geq minsupp\}$
$k \leftarrow k + 1$
end
$F = \bigcup_k F_k$

Örnek: Aşağıda verilen veritabanı ve $MinSupp = 2$ için sık öğesetlerini bulunuz.

TID	İşlem (Transaction)
T_1	$\{a, c, d\}$
T_2	$\{b, c, e\}$
T_3	$\{a, b, c, e\}$
T_4	$\{b, e\}$
T_5	$\{a, b, e\}$

İlk olarak tekli sık öğeler kümesi

$$F_1 = \{a{:}\,3, b{:}\,4, c{:}\,3, e{:}\,4\}$$

şeklinde bulunur. Daha sonra bu sık öğeler uygun şekilde birleştirilerek 2-uzunluklu aday öğesetlerinin kümesi

$$C_2 = \{ab, ac, ae, bc, be, ce\}$$

şeklinde oluşturulur. Bu adayların destek değerleri hesaplanır ve *MinSupp*=2 eşik değerini geçen öğesetleri sık öğesetleri olarak

$$F_2 = \{ab: 2, ac: 2, ae: 2, bc: 2, be: 4, ce: 2\}$$

şeklinde F_2 kümesine kaydedilir. Daha sonra F_2 kümesindeki öğesetleri uygun şekilde birleştirilerek 3-uzunluklu aday öğesetlerinin kümesi

$$C_3 = \{abc, abe, ace, bce\}$$

şeklinde oluşturulur. Bu adayların destek değerleri hesaplanır ve *MinSupp*=2 eşik değerini geçen öğesetleri sık öğesetleri olarak

$$F_3 = \{abe: 2, bce: 2\}$$

şeklinde F_3 kümesine kaydedilir. Dikkat edilirse, abc ve ace öğsetleri eşik değerini geçemediğiçin F_3 kümesinde yer almazlar. Daha sonra F_3 kümesindeki öğesetleri uygun şekilde birleştirilerek 4-uzunluklu aday öğesetlerinin kümesi

$$C_4 = \{\quad\}$$

şeklinde oluşturulur. Bu noktada, artık 4-uzunluklu aday oluşturulamayacağı için, algoritma burada sonlandırılır. Böylece,

$$F = \bigcup_k F_k = F_1 \cup F_2 \cup F_3 = \{a: 3, b: 4, c: 3, e: 4, ab: 2, ac: 2, ae: 2, bc: 2, be: 4, ce: 2, abe: 2, bce: 2\}$$

elde edilir. Tablo halinde de görülebilir:

	Öğeseti	**AbsSupp**		**Öğeseti**	**AbsSupp**
#1	$\{a\}$	3	#7	$\{a, e\}$	2
#2	$\{b\}$	4	#8	$\{b, c\}$	2
#3	$\{c\}$	3	#9	$\{b, e\}$	4
#4	$\{e\}$	4	#10	$\{c, e\}$	3
#5	$\{a, b\}$	2	#11	$\{a, b, e\}$	2
#6	$\{a, c\}$	2	#12	$\{b, c, e\}$	2

Başka bir örnekle devam edelim.

Örnek: Aşağıda verilen veritabanı ve *MinSupp*=2 için sık öğesetlerini bulunuz.

TID	**İşlem (Transaction)**
T_1	$\{b, a, g, e\}$
T_2	$\{c, b, e, f\}$
T_3	$\{f, a, e\}$

T_4	$\{b, f, g\}$
T_5	$\{b, a, d, e\}$
T_6	$\{a, f, g\}$
T_7	$\{d, a, b, e, f, g\}$
T_8	$\{a, b, e, f\}$
T_9	$\{b, c, d, f\}$
T_{10}	$\{g, a, e, f\}$
T_{11}	$\{e, b, d, f\}$

İlk olarak tekli sık öğeler kümesi

$$F_1 = \{a{:}\,7, b{:}\,8, c{:}\,2, d{:}\,4, e{:}\,8, f{:}\,9, g{:}\,5\}$$

şeklinde bulunur. Daha sonra bu sık öğeler uygun şekilde birleştirilerek 2-uzunluklu aday öğesetlerinin kümesi

$$C_2 = \{ab, ac, ad, ae, af, ag, bc, bd, be, bf, bg, cd, ce, cf, cg, de, df, dg, ef, eg, fg\}$$

şeklinde oluşturulur. Bu adayların destek değerleri hesaplanır ve *MinSupp*=2 eşik değerini geçen öğesetleri sık öğesetleri olarak

$$F_2 = \begin{Bmatrix} ab{:}\,4, ad{:}\,2, ae{:}\,6, af{:}\,5, ag{:}\,4, bc{:}\,2, bd{:}\,4, be{:}\,6, \\ bf{:}\,6, bg{:}\,3, cf{:}\,2, de{:}\,3, df{:}\,3, ef{:}\,6, eg{:}\,3, fg{:}\,4 \end{Bmatrix}$$

şeklinde F_2 kümesine kaydedilir. Dikkat edilirse, ac, cd, ce, cg, dg öğesetleri eşik değerini geçemediği için F_2 kümesinde yer almaz. Daha sonra F_2 kümesindeki öğesetleri uygun şekilde birleştirilerek 3-uzunluklu aday öğesetlerinin kümesi

$$C_3 = \begin{Bmatrix} abc, abd, abe, abf, abg, ade, adf, aef, aeg, \\ afg, bcf, bde, bdf, bef, beg, bfg, cfg, def, deg, dfg, efg \end{Bmatrix}$$

şeklinde oluşturulur. Bu adayların destek değerleri hesaplanır ve *MinSupp*=2 eşik değerini geçen öğesetleri sık öğesetleri olarak

$$F_3 = \begin{Bmatrix} abd{:}\,2, abe{:}\,4, abf{:}\,2, abg{:}\,2, ade{:}\,2, aef{:}\,4, aeg{:}\,3, afg{:}\,3, \\ bcf{:}\,2, bde{:}\,3, bdf{:}\,3, bef{:}\,4, beg{:}\,2, bfg{:}\,2, def{:}\,2, efg{:}\,2 \end{Bmatrix}$$

şeklinde F_3 kümesine kaydedilir. Dikkat edilirse, abc, adf, cfg, deg, dfg, öğesetleri eşik değerini geçemediği için F_3 kümesinde yer almaz. Daha sonra F_3 kümesindeki öğesetleri uygun şekilde birleştirilerek 4-uzunluklu aday öğesetlerinin kümesi

$$C_4 = \{abde, abdf, abef, abeg, abfg, adef, aefg, bdef, befg, defg\}$$

şeklinde oluşturulur. Bu adayların destek değerleri hesaplanır ve *MinSupp*=2 eşik değerini geçen öğesetleri sık öğesetleri olarak

$$F_4 = \{abde{:}\,2, abef{:}\,2, abeg{:}\,2, aefg{:}\,2, bdef{:}\,2\}$$

şeklinde F_4 kümesine kaydedilir. Dikkat edilirse, $abdf, abfg, adef, befg, defg$ öğesetleri eşik değerini geçemediği için F_4 kümesinde yer almaz. Daha sonra F_4 kümesindeki öğesetleri uygun şekilde birleştirilerek 5-uzunluklu aday öğesetlerinin kümesi

$$C_5 = \{abdef\}$$

şeklinde oluşturulur. Bu adayların destek değerleri hesaplanır ve *MinSupp*=2 eşik değerini geçen öğesetleri sık öğesetleri olarak

$$F_5 = \{\,\}$$

Bulunur. Bu noktada, artık 6-uzunluklu aday oluşturulamayacağı için, algoritma burada sonlandırılır. Böylece,

$$\begin{aligned} F &= \bigcup_k F_k \\ &= F_1 \cup F_2 \cup F_3 \cup F_4 \\ &= \left\{ \begin{array}{c} a: 7, b: 8, c: 2, d: 4, e: 8, f: 9, g: 5, \\ ab: 4, ad: 2, ae: 6, af: 5, ag: 4, bc: 2, bd: 4, be: 6, \\ bf: 6, bg: 3, cf: 2, de: 3, df: 3, ef: 6, eg: 3, fg: 4 \\ abd: 2, abe: 4, abf: 2, abg: 2, ade: 2, aef: 4, aeg: 3, afg: 3, bcf: 2, \\ bde: 3, bdf: 3, bef: 4, beg: 2, bfg: 2, def: 2, efg: 2 \\ abde: 2, abef: 2, abeg: 2, aefg: 2, bdef: 2 \end{array} \right\} \end{aligned}$$

elde edilir. Tablo halinde de görülebilir:

	Öğeseti	**AbsSupp**		**Öğeseti**	**AbsSupp**
#1	$\{a\}$	7	#23	$\{f, g\}$	4
#2	$\{b\}$	8	#24	$\{a, b, d\}$	2
#3	$\{c\}$	2	#25	$\{a, b, e\}$	4
#4	$\{d\}$	4	#26	$\{a, b, f\}$	2
#5	$\{e\}$	8	#27	$\{a, b, g\}$	2
#6	$\{f\}$	9	#28	$\{a, d, e\}$	2
#7	$\{g\}$	5	#29	$\{a, e, f\}$	4
#8	$\{a, b\}$	4	#30	$\{a, e, g\}$	3
#9	$\{a, d\}$	2	#31	$\{a, f, g\}$	3
#10	$\{a, e\}$	6	#32	$\{b, c, f\}$	2
#11	$\{a, f\}$	5	#33	$\{b, d, e\}$	3
#12	$\{a, g\}$	4	#34	$\{b, d, f\}$	3
#13	$\{b, c\}$	2	#35	$\{b, e, f\}$	4
#14	$\{b, d\}$	4	#36	$\{b, e, g\}$	2
#15	$\{b, e\}$	6	#37	$\{b, f, g\}$	2
#16	$\{b, f\}$	6	#38	$\{d, e, f\}$	2
#17	$\{b, g\}$	3	#39	$\{e, f, g\}$	2
#18	$\{c, f\}$	2	#40	$\{a, b, d, e\}$	2
#19	$\{d, e\}$	3	#41	$\{a, b, e, f\}$	2
#20	$\{d, f\}$	3	#42	$\{a, b, e, g\}$	2
#21	$\{e, f\}$	6	#43	$\{a, e, f, g\}$	2
#22	$\{e, g\}$	3	#44	$\{b, d, e, f\}$	2

Apriori algoritması, kendinden sonra gelen pek çok algoritmaya ilham kaynağı olmuş önemli bir algoritmadır. Gerçeklemesi kolaydır. Ancak, pek çok dezavantajı da içerisinde barındırmaktadır. Örneğin, Apriori algoritması, aday öğeseti oluştururken veritabanına bakmaz; dolayısıyla, veritabanında bulunmayan aday öğesetleri de oluşturabilir. Bu da çok büyük zaman ve enerji kaybına yol açabilir. Diğer bir dezavantajı da, aday öğesetlerinin destek değerlerini bulurken veritabanını defalarca taraması gerekmektedir ki bu da işlem maliyeti açısından yükü artırmaktadır. Başka bir dezavantajı ise, breadth-first search arama yaklaşımının hafıza gereksinimi açısından kullanışlı olmamasıdır. En kötü durumda bile, tüm k ve $k-1$ uzunluklu öğesetlerini hafızada tutmak zorundadır. Bir algoritmanın çalıştırılması için geçen süreyi tanımlayan hesaplama karmaşıklığını ifade eden z*aman karmaşıklığı* (*time complexity*) açısından bakıldığında, Apriori algoritmasının *time complexity* değeri, m farklı öğe sayısı ve N işlem sayısı olmak üzere, $O(m^2N)$ olarak verilmiştir.

3.3 ECLAT Algoritması [2]

ECLAT (**E**quivalence **CLA**ss **T**ransformation) algoritması *dikey veritabanı* (*vertical database*) ve hafızayı daha iyi kullanmak için depth-first search yaklaşımını kullanır. Dikey veritabanında, bir öğenin bulunduğu işlemlerin listesi vardır. Örnek olarak, aşağıdaki gibi verilen bir yatay veritabanını ele alalım:

TID	İşlem (Transaction)
T_1	$\{a, c, d\}$
T_2	$\{b, c, e\}$
T_3	$\{a, b, c, e\}$
T_4	$\{b, e\}$
T_5	$\{a, b, c, e\}$

X gibi bir öğeseti için bu öğeyi içeren işlemlerin listesine X öğesetinin *işlem-listesi* (TID-list) denir ve $tid(X)$ ile gösterilir. Tablo 6.4 ile verilen veritabanının dikey gösterilimi Tablo 5'deki gibidir.

X	$tid(X)$
$\{a\}$	$\{T_1, T_3, T_5\}$
$\{b\}$	$\{T_2, T_3, T_4, T_5\}$
$\{c\}$	$\{T_1, T_2, T_3, T_5\}$
$\{d\}$	$\{T_1\}$
$\{e\}$	$\{T_2, T_3, T_4, T_5\}$

Bu dikey veritabanı, verilen yatay veritabanı sadece bir kez taranarak elde edilebilir. Tabi, bunun tersi de mümkündür, yani, dikey veritabanından yatay veritabanına geçmek de mümkündür. Dikey veritabanı yaklaşımı, iki özelliğinden dolayı veri madenciliğinden çokça kullanılmaktadır: İlki, X ve Y gibi herhangi iki öğeseti için, bu ikisinin birleşiminden oluşan $X \cup Y$ öğeseti, veritabanı tekrar taranmadan, $tid(X)$ ve $tid(Y)$ kümelerinin kesişiminden bulunabilir, yani,

$$tid(X \cup Y) = tid(X) \cap tid(Y).$$

Örnek olarak, $\{a\}$ ve $\{b\}$ öğelerinden oluşan $\{a, b\}$ öğesetinin işlem-listesi şu şekilde bulunur:

$$tid(\{a\} \cup \{b\}) = tid(\{b\}) \cap tid(\{b\}) = \{T_3, T_5\}$$

İkinci özellik, X öğesetinin destek değeri, veritabanı tekrar taranmadan, işlem-listesi olan $tid(X)$ kullanılarak şu şekilde bulunabilir:

$$sup(X) = |tid(X)|.$$

Örnek olarak, $\{a, b\}$ öğesetinin destek değeri şu şekilde bulunur:

$$sup(\{a, b\}) = |tid(\{a, b\})| = 2.$$

Böylece, bu iki özelliği kullanarak ECLAT algoritması gibi dikey veritabanı kullanan algoritmalar veritabanını sadece bir kez tarayarak ilk işlem-listelerini oluştururlar. Artık bundan sonra, aday öğeseti oluşturma ve destek hesabı tamamen bu dikey veritabanından gerçekleştirilir. ECLAT algoritmasının sözde-kodu aşağıda verilmiştir.

Algoritma. ECLAT Algoritması

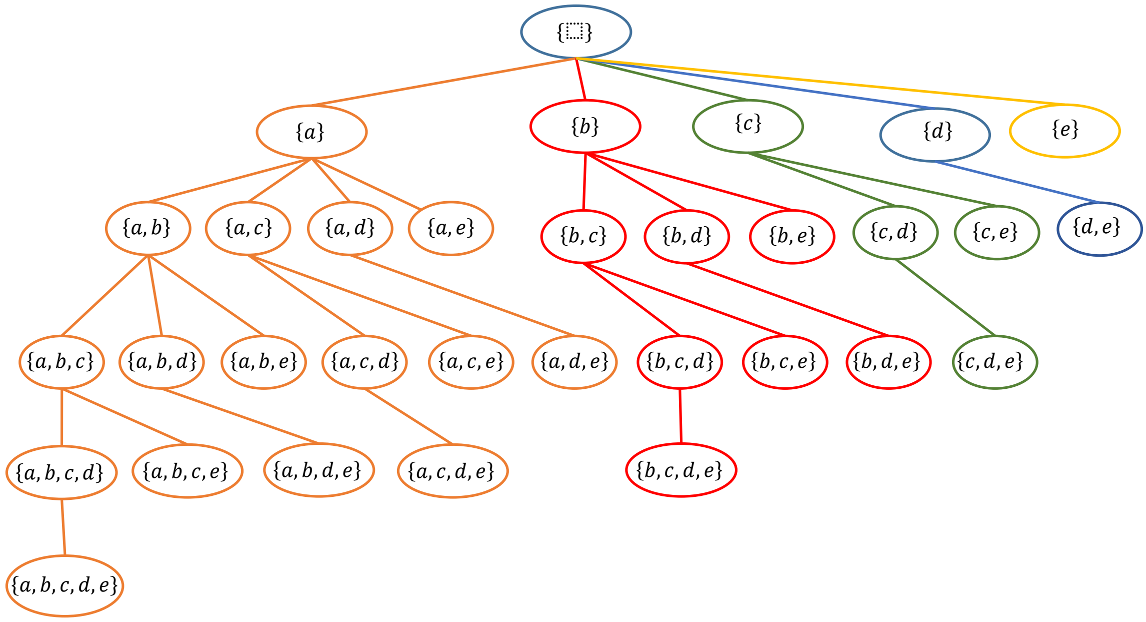
Girdiler: R: Dikey işlem veritabanı, $MinSupp$: eşik değeri
Çıktı: F Sık öğesetleri ve destek değerleri

1	**for** each itemset $X \in R$ such that $\|tid(X)\| \geq MinSupp$	
2	X öğsetini çıktı olarak gönder	*X bir sık öğesetidir*
3	$E = \emptyset$	
4	**for** each itemset $Y \in R$ sharing all but the last item with X	*X ile sonuncusu hariç tüm öğeleri aynı olan Y öğesetleri*
5	$tid(X \cup Y) = tid(X) \cap tid(Y)$	*$X \cup Y$'nin işlem-listesini bul*
6	**if** $\|tid(X \cup Y)\| \geq MinSupp$ **then**	*Eğer $X \cup Y$ sık öğe ise*
7	$E = E \cup \{X \cup Y\}$	*$X \cup Y$ töğesetini X'in sık öğeseti listesi E'ye ekle*
8	**end**	
9	ECLAT($E, MinSupp$)	*ECLAT algoritmasını E için yinelemeli olarak çağır (Recursive call using E)*
10	**end**	
11	**end**	

ECLAT agolritmasının girişi, dikey veritabanı R (Tablo'da görüldüğü gibi) ve $MinSupp$ eşik değeridir. ECLAT algoritması, dikey veritabanı R'nin içerisinde sık olan her bir X öğesetini dikkate alarak bir döngü gerçekleştirir (2–10 satırları). X öğeseti ilk çıktıdır. Daha sonra, X öğesetine bir öğe ekleyerek genişletilerek sık öğeseti bulmak için arama yapılır. Bu şu şekilde yapılır: R içerisinde, X öğesetiyle biri hariç tüm elemanları aynı olan her bir Y öğesetini birleştirerek $X \cup Y$ öğeseti elde edilir (4–10 satırları). Örnek olarak, eğer $X = \{a\}$ ise, o zaman ECLAT X öğesetini $\{b\}$, $\{c\}$, $\{d\}$, ve $\{e\}$ öğesetleriyle birleştirerek sırasıyla $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, ve $\{a, e\}$ öğesetlerini oluşturacaktır. Bu işlem esnasında, $X \cup Y$'nin işlem-listesi $tid(X \cup Y) = tid(X) \cap tid(Y)$ ile bulunur (satır 6). Eğer $X \cup Y$ sık bir öğesetiyse o zaman, X öğesetinin genişletilmiş sık öğeseti kümesi olan F'ye eklenir (6-7 satırları). Ardından, ECLAT algoritması, $X \cup Y$'nin tüm uzantılarını bulmak için E kümesi ile birlikte kendi kendini çağırır. Bu döngü, R'nin içindeki tüm öğesetleri için tekrarlanır. Algoritma bittiğinde elde edilen sık öğesetleri ve onların destek değerleri çıkışı oluşturur. ECLAT algoritmasının arama uzayı aşağıdaki şekilde görülmektedir. Algoritma, öğe-kümelerini şu sıra ile oluşturur:

$\{a\}$, $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, $\{a, e\}$, $\{a, b, c\}$, $\{a, b, d\}$, $\{a, b, e\}$, $\{a, c, d\}$, $\{a, c, e\}$, $\{a, d, e\}$, $\{a, b, c, d\}$, $\{a, b, c, e\}$, $\{a, b, d, e\}$, $\{a, c, d, e\}$, $\{a, b, c, d, e\}$, $\{b\}$, $\{b, c\}$, $\{b, d\}$, $\{b, e\}$, $\{b, c, d\}$, $\{b, c, e\}$, $\{b, d, e\}$, $\{b, c, d, e\}$, $\{c\}$, $\{c, d\}$, $\{c, e\}$, $\{c, d, e\}$, $\{d\}$, $\{d, e\}$, $\{e\}$.

ECLAT algoritması çıktıları depth-first search yaklaşımındaki sıraya göre ürettiği için bir depth-first search algoritması olarak görülebilir. Veritabanını çok sayıda taramadığı için Apriori



Şekil. ECLAT algoritması için $I = \{a, b, c, d, e\}$ kümesinin depth-first search arama uzayı

algoritmasından her zaman daha hızlıdır. Ancak, ECLAT algoritmasının bazı dezavantajları vardır. ECLAT, aday öğesetlerini bulurken veritabanını taramadığı için veritabanında bulunmayan öğeseti adayları oluşturabilir ki bu da zaman kaybına yol açar. Diğer bir dezavantajı, her ne kadar işlem-listeleri (*tid-list*) faydalı olsa da, özellikle yoğun (tüm öğelerin hemen hemen her işlemde yer aldığı veritabanları) veritabanları için hafızada çok yer tutar. Yine de, işlem-listelerinin (*tid-list*) boyutunu azaltan yapıların literatürde önerildiğini söylemekte fayda vardır. İşlem-listelerini bit-vektörleri şeklinde kodlayarak boyutları azaltılabilir ki böylece işlem hızı artar. İşlem-listeleri, ayrıca, Apriori-TID gibi breadth-first search algoritmalarında da kullanılabilir.

3.4 H-mine Algoritması [3]

Aday oluşturma ilkesine dayanan Apriori ve ECLAT algoritmaları, sık olmayan öğesetlerini de aramakta, bu da zaman kaybına yol açmaktadır. Buna bir çare olarak H-mine algoritması önerilmiştir. H-mine, Aramayı depth-first (derinlik öncelikli) tarzda yapan, yatay veritabanı kullanan bir algoritmadır. Apriori ve ECLAT'ın aksine, aday oluşturmaz, bir önekin (prefix) project ettiği veritabanında destek değeri *MinSupp* değerinden büyük olan öğelerle aramaya devam eder. Bu özelliğinden dolayı, Apriori ve ECLAT algoritmalarına göre daha hızlıdır. Aşağıdaki veritabanını ele alalım:

TID	İşlem (Transaction)
T_1	$\{ c, d, e, f, g, i\}$
T_2	$\{a, c, d, e, m\}$
T_3	$\{a, b, d, e, g, k\}$
T_4	$\{a, c, d, h\}$

Sadece dört işlemden oluşan bu veritabanı için *AbsMinSupp* değeri 2 olarak belirlensin. *Apriori* özelliğine göre, sadece sık öğelerin dikkate alınması gerekmektedir. Dolayısıyla, veritabanı bir kez taranarak sık öğeler sırasıyla, a:3, c:3, d:4, e:3, g:2 şeklinde bulunur, burada : işaretinden sonraki rakam o öğenin veritabanındaki görülme sayısıdır. Sadece sık öğeler dikkate alındığında, veritabanı aşağıdaki hale gelir:

TID	İşlem (Transaction)
T_1	$\{ c, d, e, g\}$
T_2	$\{a, c, d, e\}$
T_3	$\{a, d, e, g\}$
T_4	$\{a, c, d\}$

Bu sık öğeler alfabetik sıraya dizilirse, $F_{list} = a, c, d, e, g$ şeklinde bir F_{list} (sık öğeler listesi) oluşturulur ve verilen *AbsMinSupp* =2 değeri için tüm sık öğesetleri beş ana gruba ayrılabilir. Bu gruplar şu şekildedir:

a- ile başlayan sık öğesetleri,

c- ile başlayan ancak a öğesi içermeyen sık öğesetleri,

d- ile başlayan ancak a ve c öğelerini içermeyen sık öğesetleri,

e- ile başlayan ancak a, c ve d öğelerini içermeyen sık öğesetleri

g- ile başlayan ancak a, c, d ve e öğelerini içermeyen sık öğesetleri.

Böylece, F_{list} içerisindeki sık öğeler, bu öğelerin destek değerleri (veritabanındaki görülme sayısı) ve her bir öğeyi veritabanında öğenin bulunduğu ilgili satırlara bağlayan bağlantıların (hyperlink) bulunduğu bir başlık (header) tablosu şu şekilde elde edilir:

H header tablosu				
a	c	d	e	g
3	3	4	3	2

c	d	e	g
a	c	d	e
a	d	e	g
a	c	d	

Artık sık öğesetlerinin bulunmasına a öğesi ile başlanabilir. a öğesi ile başlayan sık öğesetlerini bulmak için gerekli olan a-projected veritabanını oluşturmak için a öğesine ilişkin header tablosu H_a oluşturulmalıdır. H_a tablosunun içinde, a dışındaki tüm sık öğeler tüm bileşenleriyle birlikte bulunmaktadır. Buradaki destek değerleri, sadece a-projected veritabandaki sayılar dikkate alınarak belirlenir. a-projected veritabanı bir kez tarandığında, lokal olarak sık olan öğeler şu şekilde belirlenir: c:2, d:3, e:2, burada g öğesinin lokal destek değeri 1 olduğu için g öğesi elenmiştir. Bunun sonucunda, sık öğesetleri şu şekilde oluşur: ac:2, ad:3, ae:2.

H_a header tablosu				
	c	d	e	g
	2	3	2	1

c	d	e	g
a	c	d	e
a	d	e	g
a	c	d	

Benzer işleme ac-projected veritabanı ile devam edelim.

H_{ac} header tablosu				
	c	d	e	g
		2	1	0

c	d	e	g
a	c	d	e
a	d	e	g
a	c	d	

ac-projected veritabanındaki lokal sık öğe sadece d öğesi olup destek değeri 2'dir, ve acd bir sık öğesetidir. O zaman, işleme acd-projected veritabanı ile devam edelim.

H_{acd} header tablosu				
	c	d	e	g
			0	0

c	d	e	g
a	c	d	e
a	d	e	g
a	c	d	

acd-projected veritabanında lokal sık öğe bulunmadığından, sık örüntü aramaya benzer şekilde ad-projected veritabanı ile devam edilir. Bunu yaparken c öğesi dikkate alınmaz.

H_{ad} header tablosu				
	c	d	e	g
			2	1

c	d	e	g
a	c	d	e
a	d	e	g
a	c	d	

ad-projected veritabanındaki lokal sık öğe sadece e öğesi olup ade öğeseti bir sık öğesetidir ve destek değeri 2'dir ve ad- ile başlayan başka bir öğe kalmadığından ad-projected veritabanından son olarak ae-projected veritabanına geçilir. ae-projected veritabanda ae- önekini takip edebilecek lokal olarak sık bir öğe bulunmadığından, a- ile başlayan tüm sık öğesetleri bulunmuş olur. Sonuç olarak, a- ile başlayan tüm öğesetleri sırasıyla şu şekilde bulunmuş olur: a: 3, ac:2, acd:2, ad:3, ade:2, ae:2. Bu işleme, benzer şekilde, c-, ile başlayan sık öğesetlerinin bulunmasıyla devam edilir ancak bunu yaparken artık a öğesi dikkate alınmaz. Benzer şekilde, d- ile başlayan sık öğesetlerinin bulunurken a ve c öğeleri dikkate alınmaz; e- ile başlayan sık öğesetlerinin bulunurken a, c ve d öğeleri dikkate alınmaz.

H-mine algoritması gerçeklenirken, veritabanının gösterilimi çok önemlidir. Çünkü, belli bir öğe önek (prefix) olarak belirlendiğinde, örneğin a öğesi önek olsun, a- ile başlayan sık öğesetlerini bulmak için veritabanında a'nın yer aldığı satırların bulunması gerekir. Ayrıca, bu satırlar bulunduktan sonra da, sadece bu satırlarda yer alan diğer öğelerin sayılarının da bulunması gerekir. Bunun için veritabanında her bir verinin gösterilimi hız ve hafıza kullanımı gibi performans ölçütlerine etki eder. H-mine algoritmasını etkin bir şekilde gerçeklemek için en uygun gösterilim şekli her bir öğenin ilgili işlemdeki varlığı veya yokluğuna göre `1` veya `0` kullanılan ikili (binary) gösterilim şeklidir. Örneklerde kullanılan veritabanını ele alalım. Her bir öğeyi, işlemdeki varlığına göre, eğer varsa `1`, yoksa `0` şeklinde ifade edersek, örneğin $\{c, d, e, g\}$ işlemini `01111` ile temsil ederiz. Diğer işlemler için de aynı gösterilimi uygularsak, tablonun üçüncü sütunundaki gibi bir veritabanı elde edilir. Veritabanı, aslında ikili sayılardan oluşan bir matristir. Bu matrisin ilk sütunu a, ikinci sütunu c, üçüncü sütunu d, dördüncü sütunu e ve son sütunu da g öğesinin işlemlerdeki durumunu göstermektedir. Bu gösterilim kullanılarak, örneğin a- ile başlayan sık öğesetlerini bulmak için veritabanında a'nın yer aldığı satırların bulunması daha da kolaylaşır, çünkü bunun için matrisin ilk sütunundaki `1` elemanlarının indislerinin bulunması yeterlidir. Bu satırlarda yer alan öğelerin destek değerlerinin bulunması, bu satırlardaki değerlerin toplanması ile bulunabilir.

TID	İşlem (Transaction)	İkili Gösterimli İşlem				
		a	c	d	e	g
T_1	$\{c, d, e, g\}$	0	1	1	1	1
T_2	$\{a, c, d, e\}$	1	1	1	1	0
T_3	$\{a, d, e, g\}$	1	0	1	1	1
T_4	$\{a, c, d\}$	1	1	1	0	0

Örnek olarak, a- ile başlayan sık öğesetlerinin bulundauğu satırların indisleri sırasıyla, 2, 3 ve 4 şeklindedir. Bu satırlarda yer alan öğelerin destek sayıları sırasıyla, c:2, d:3, e:2 ve g:1 şeklindedir.

	a	c	d	e	g
1	0	1	1	1	1
2	1	1	1	1	0
3	1	0	1	1	1
4	1	1	1	0	0

Dolayısıyla, H-mine algoritmasını etkin bir şekilde gerçeklemek için ikili elemanlardan oluşan matrix gösterilimi oldukça uygundur.

3.5 FPtree Algoritması [4]

FPtree algoritması, Apriori, ECLAT ve H-mine algoritmalarından farklı olarak, veritabanını sadece iki kez tarar. Bu algoritmalar, veritabanını çok kereler tarmaktadır ve veritabanındaki işlem sayısı arttıkça, algoritmaların sonucu üretme zamanı da artmaktadır. Buna bir çare olarak veritabanını sadece iki kez tarayan FPtree algoritması önerilmiştir. FPtree, veritabanındaki tüm verileri kayıpsız bir şekilde bir ağaç yapısına aktarır ve sık öğeseti çıkarımını bu ağaç üzerinden yapar. Veritabanında, aynı öğelere sahip işlemler ağaçta aynı kolda yer aldığı için hem kompakt bir yapıya sahiptir hem de sık öğeseti çıkarımı daha hızlı olmaktadır. FPtree algoritması iki aşamadan oluşur: (1) Veritabanından ağacın oluşturulması, (2) Oluşturulan ağaçtan sık öğesetlerinin elde edilmesi. Bu aşamaları, bir örnek üzerinden görelim: Aşağıdaki gibi bir işlem veritabanı (Transaction Database - TD) ele alalım:

İşlem	İşlem
#1	{A, B, C}
#2	{A, E}
#3	{B, D, E}
#4	{A, D, E}
#5	{A, B, C}
#6	{A, E}
#7	{A, C, D}
#8	{B, C, E}
#9	{A, C, D, E}
#10	{A, B, C, E}

Bu şekilde verilen bir veritabanını, uygun bir ağaç ile temsil edebilmek için ilk olarak, ağacın daha kompakt bir yapıda olmasını sağlamak için TD yeniden düzenlenir. Bunu sağlamak için, A, B, C, D ve E öğelerinin destek (support) değerleri, yani her bir öğenin içerisinde bulunduğu işlem sayısı bulunarak hesaplanırsa, aşağıdaki tablodaki gibi bulunur:

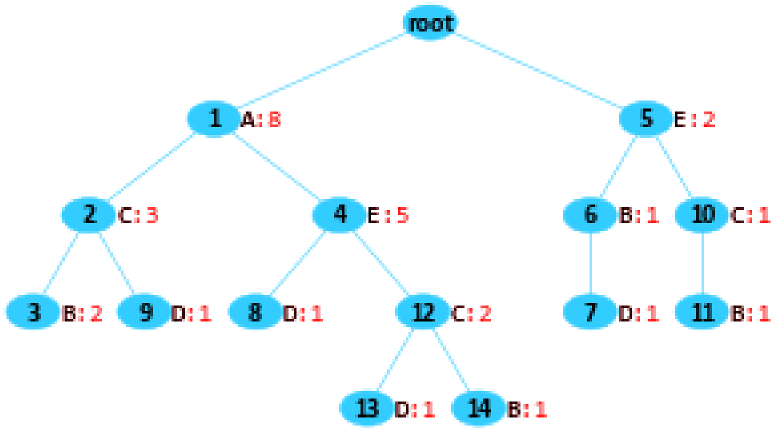
Öğe	Destek
A	8
B	5
C	6
D	4
E	7

Bu öğeler, destek değerleri büyükten küçüğe doğru A, E, C, B ve D şeklinde sıralanırlar. Öğelerin öncelik sırası A, E, C, B ve D şeklindedir. TD'deki her bir işlem bu sıraya göre yeniden düzenlenirse, yeni TD şu hale gelir:

İşlem	İşlem
#1	{A, C, B}
#2	{A, E}
#3	{E, B, D}

#4	{A, E, D}
#5	{A, C, B}
#6	{A, E}
#7	{A, C, D}
#8	{E, C, B}
#9	{A, E, C, D}
#10	{A, E, C, B}

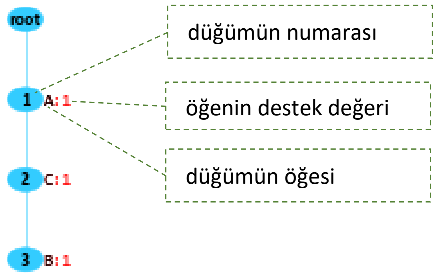
Esasında bu iki TD arasında hiç bir fark yoktur. Bu yeni TD, aşağıdaki ağaç ile temsil edilebilmektedir.



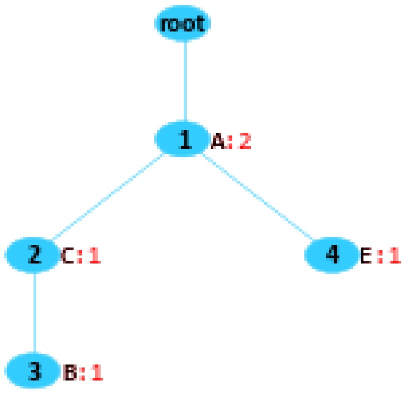
Ağacın en tepesinde bir kök (root) düğümü, en altlarda ise yaprak düğümleri bulunmaktadır. Buradaki her bir düğüm, o düğüme ilişkin öğeyi, ardışık düğümlerden oluşan kollar ise öğesetlerini göstermektedir. Öğelerin yanında kırmızı renkle yazılmış olan tamsayılar ise o öğenin o kolda kaç kere göründüğünü göstermektedir. Düğümlerin içindeki rakamlar ise sadece düğümlere verilen birer düğüm numarasıdır. Örnek olarak, 1-2-3 nolu düğümlerinden oluşan koldaki öğeseti {A, C, B}'dir, bu öğesetinin görülme sayısı, en uçtaki öğenin görülme sayısına eşittir, yani 2'dir. Gerçekten de {A, C, B} öğeseti veritabanında 2 işlemde bulunmaktadır. Dolayısıyla, veritabanının #1 ve #5 nolu satırlarında bulunan {A, C, B} öğeseti, ağaçta 1-2-3 nolu düğümlerinden oluşan kol ile temsil edilmektedir. Veritabanındaki diğer işlemler de benzer şekilde ağaç üzerinde görülebilir. Sonuç olarak, verilen bir işlem veritabanı, bir ağaç ile temsil edilebilmektadir. Aynı işlem veritabanını temsil eden birbirinden farklı çok sayıda ağaç olabilir ancak, başlangıçta yapılan sıralama sayesinde en az sayıda düğüme sahip ağaç elde edilmiştir. Şimdi bu ağacın nasıl oluşturulduğuna daha yakından bakalım.

3.8.1 Ağaç Oluşturma

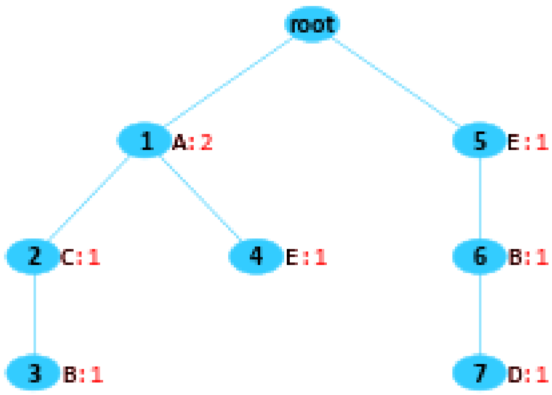
TD'deki işlemler, işlem numaralarına göre sırayla alınarak ağaç oluşturulur. İlk olarak #1 nolu işlem, yani, {A, C, B} işlemi alınırsa ağacın ilk dalı şu şekilde oluşur:



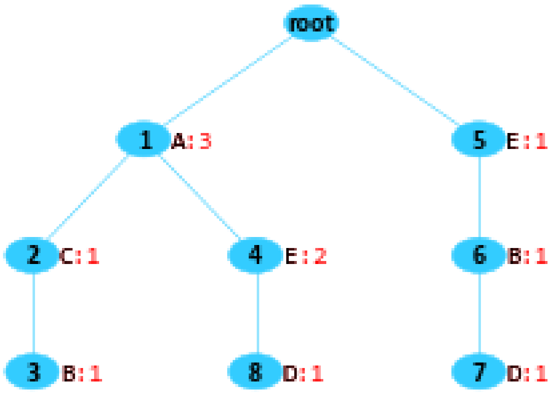
Bu ağaçta, en tepede bir kök (root) düğümü vardır. İşlem {A, C, B} şeklinde olduğu için, kök düğümün altında, A düğümü, onun altında C düğümü, onun altında da B düğümü oluşur. Yukarıdaki şekilde görüldüğü gibi, düğümde, dairenin içerisinde yer alan rakam, düğümün oluşturulma sırasını gösterir, düğümün yanındaki harf, düğüme ilişkin öğedir, öğelerin yanında ":" sembolünden sonra gelen rakamlar düğümün destek değerlerini göstermektedir. Her bir öğe şimdilik sadece bir kez gözüktüğü için karşılarında 1 rakamı vardır. #1 nolu işlem bu şekilde ağaca işlendikten sonra, #2 nolu işlem olan {A, E} işlemine geçilir. Bu işlemin ilk elemanı A öğesidir ve ağaçta A ile başlayan bir dal olduğu için yeni bir dal oluşturulmaz, sadece A öğesinin destek değeri 1 artırılır. A öğesinden sonra E öğesi gelmektedir ancak mevcut ağaçta A öğesinden sonra E öğesi olmadığı için E öğesini içeren yeni bir dal oluşturularak destek değeri 1 olarak belirlenir ve ağaç aşağıdaki hale gelir:



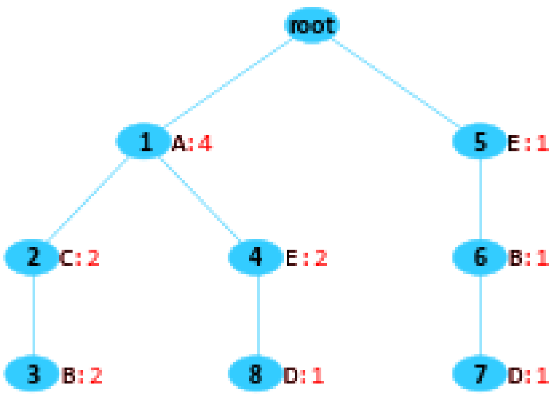
Ağacı oluşturmaya, benzer şeklide, #3 nolu işlem olan {E, B, D} işlemi ile devam edilir. Bu işlem E öğesi ile başlamaktadır ancak ağaçta E ile başlayan bir dal bulunmadığı için bu işlemin tamamı yeni bir dal olarak ağaca eklenir ve ağaç aşağıdaki hale gelir:



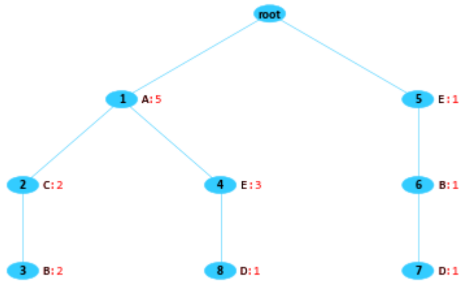
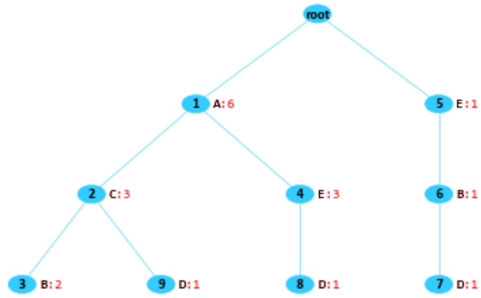
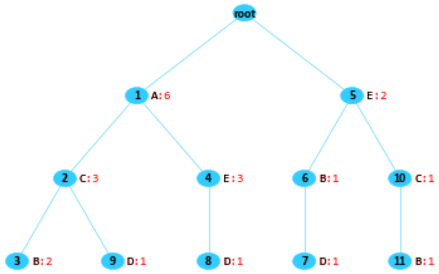
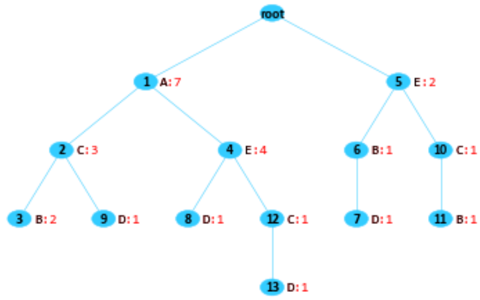
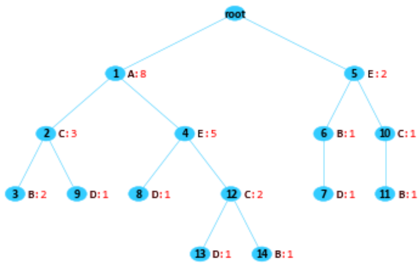
Ağacı oluşturmaya #4 nolu işlem olan {A, E, D} işlemiyle devam edilirse, ağaç şu hale gelir:



Görüldüğü gibi, {A, E} ile başlayan bir dal olduğu için bu daldaki düğümlerin destek değerleri 1 artırılmıştır. Sadece, D öğesi için E'nin altına bir düğüm eklenmiştir. Ağacı oluşturmaya #5 nolu işlem olan {A, C, B} işlemi ile devam edilirse, ağaçta önceden bir {A, C, B} dalı olduğu için yeni bir dal ya da düğüm eklenmez, sadece {A, C, B} dalındaki düğümlerin destek değerleri 1 artırılır ve ağaç şu hale gelir:



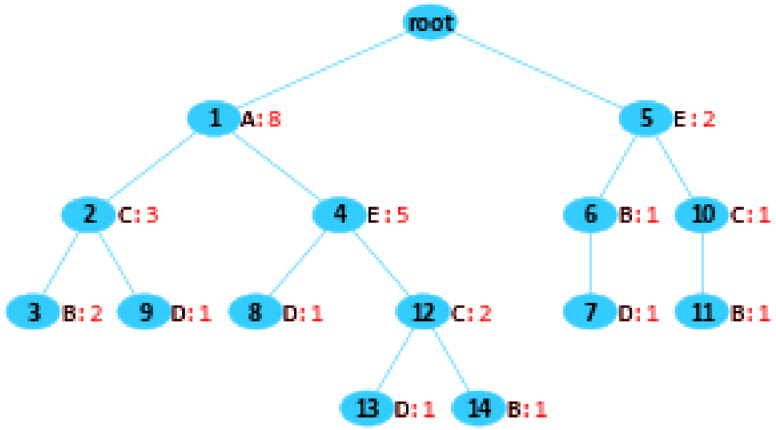
#6 nolu işlemden itibaren, ağacı oluşturma işine bu şekilde devam edildiğinde, ağacın oluşum evreleri aşağıdaki tablodaki gibi olur:

Eklenen İşlem No.	Eklenen İşlem	Ağaç
#6	{A, E}	
#7	{A, C, D}	
#8	{E, C, B}	
#9	{A, E, C, D}	
#10	{A, E, C, B}	

Böylece,

İşlem No.	İşlem
#1	{A, C, B}
#2	{A, E}
#3	{E, B, D}
#4	{A, E, D}
#5	{A, C, B}
#6	{A, E}
#7	{A, C, D}
#8	{E, C, B}
#9	{A, E, C, D}
#10	{A, E, C, B}

şeklinde verilen bir işlem veritabanı (transaction database - TD), aşağıdaki gibi bir ağaç ile temsil edilmiş olur.

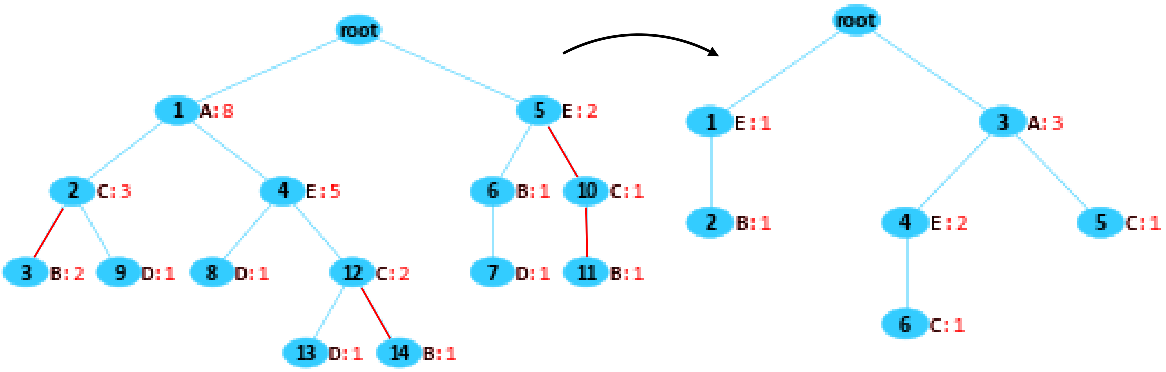


Artık bu ağaç kullanılarak, sık öğesetleri bulunabilir.

3.8.2. Ağaçtan Sık Öğesetlerinin Bulunması

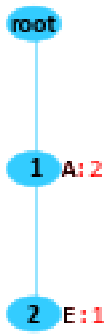
Oluşturulan ağaçtan sık öğesetlerinin bulunması için, *TopDown* ve *BottomUp* olmak üzere iki farklı yaklaşım vardır. Burada sadece *BottomUp* yaklaşımı ele alınacaktır. *BottomUp* yaklaşımı sık öğesetlerini bulmaya yapraklardan başlayarak kök düğümüne doğru ilerler. *MinSupp*=0.2 için sık öğesetlerini bulalım. Bu değer aynı zamanda *AbsMinSupp*=2 değerine eşdeğerdir. Hatırlanacağı gibi, öğelerin destek değerleri sırasıyla, A:8, E:7, C:6, B:5 ve D:4 şeklindeydi. Dolayısıyla bu tekli öğeler doğal olarak sık öğesetleridir. Bu destek değerlerine göre öğelerin öncelik sırasının A, E, C, B ve D şeklinde olduğunu hatırlayalım. Öğesetlerinin bulunmasına yapraklardan başlanacağı için ve öncelik sırası A, E, C, B, D şeklinde olduğu için ilk olarak D- ile biten sık öğesetleri bulunacaktır. Bunun için de ağaçta sadece D- ile biten dallar tutulur, geri kalan dallar ağaçtan atılarak D-sonekli ağaç (D-suffixed tree) aşağıdaki gibi oluşturulur. Orjinal ağaç ile D-sonekli ağaç karşılaştırıldığında görülecektir ki, D-sonekli ağaçta sadece yapraklarında D öğesi olan dallar bulunmaktadır. Bu dallar, aşağıdaki orjinal ağaçta mavi renkle gösterilmiştir. Yapraklarında D öğesi bulundurmayan dallar, kırmızı renk ile gösterilmiştir, D-sonekli ağaçta yer almamaktadır. Ayrıca, D-sonekli ağacın yapraklarında D öğesinin bulunmasına ya da gösterilmesine gerek yoktur, çünkü zaten ağaç D-sonekli ağaçtır, yani yapraklarının hepsinde D öğesi vardır.

orjinal ağaç | D-sonekli ağaç



D-sonekli ağaç elde edildikten sonra, bu ağaçta bulunan öğelerin destek değerlerinin sayılması işlemine geçilir. D-sonekli ağaçtaki öğelerin destek değerleri sırasıyla, A:3, E:3, C:2, B:1 şeklindedir. *AbsMinSupp* = 2 olduğu için, bu eşik değerini sadece A, E ve C öğeleri geçer; dolayısıyla buradan AD:3, ED:3, CD:2 şeklinde üç tane sık öğeseti bulunmuş olur. Aramaya, CD-sonekli ağaç ile devam edilir. CD-sonekli ağaç aşağıdaki gibidir:

CD-sonekli ağaç

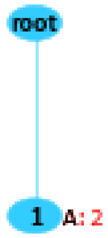


Şimdi de CD-sonekli ağaçtaki öğelerin destek değerlerini bulalım. Bu destek değerleri, A:2 ve E:1 şeklindedir. *AbsMinSupp* = 2 eşik değerinin sadece A öğesi geçtiği için, buradan sadece ACD:2 sık öğeseti gelir. Şimdi de ACD-sonekli ağacı elde edelim. ACD-sonekli ağaç, aşağıdaki şekilden de görüleceği gibi, sadece kök düğümünden ibaretttir; dolayısıyla buradan daha başka sık öğeseti gelmez.

ACD-sonekli ağac

D-sonekli ağaca devam edilirse, sırada ED-sonekli ağaç vardır. ED-sonekli ağaç, aşağıdaki gibidir.

ED-sonekli ağaç



Şimdi de ED-sonekli ağaçtaki öğelerin destek değerlerini bulalım. Bu destek değerleri, A:2 şeklindedir. Buradan sadece AED:2 sık öğeseti gelir. Aramaya, aşağıdaki gibi, AED-sonekli ağaçla devam edilecektir, ancak AED-sonekli ağaç sadece kök düğümden ibaret olduğu için buradan daha başka sık öğeseti gelmez.

AED-sonekli ağaç

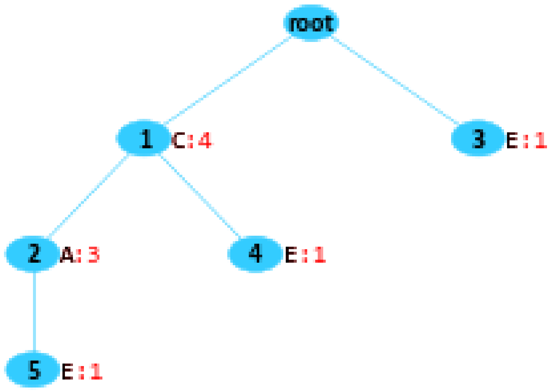
D-sonekli ağaca devam edilirse, sırada AD-sonekli ağaç vardır. AD-sonekli ağaç, aşağıdaki gibidir, ancak AD-sonekli ağaç sadece kök düğümden ibaret olduğu için buradan daha başka sık öğeseti gelmez.

AD-sonekli ağaç

Böylece, D-sonekli sık öğesetlerinin tamamı bulunmuş olur. D-sonekli sık öğesetleri şu şekildedir. D:4, CD:2, ACD:2, ED:3, AED:2, AD:3.

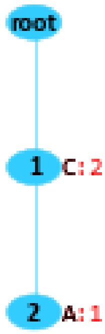
Şimdi, benzer şekilde, sıralamada bir sonraki öğe olan B öğesi ile biten sık öğesetlerini bulalım. İlk olarak B-sonekli ağaç aşağıdaki gibi elde edilir:

B-sonekli ağaç



B-sonekli ağaç elde edildikten sonra, bu ağaçta bulunan öğelerin destek değerlerinin sayılması işlemine geçilir. B-sonekli ağaçtaki öğelerin destek değerleri sırasıyla, C:4, A:3, E:3 şeklindedir. B-sonekli ağaçta sıralamanın değiştiğine dikkat edilmedilir. Orjinal ağaçta sıralama A, E, C, B, D şeklinde iken, B-sonekli ağaçta C, A, E şeklinde olmuştur. Bu sıralama tamamen o anki ağaçtaki öğelerin destek sayılarına göre belirlenmektedir. $AbsMinSupp = 2$ olduğu için, bu eşik değerini sadece C, A, ve E öğeleri geçer; dolayısıyla buradan CB:4, AB:3, EB:3 şeklinde üç tane sık öğeseti bulunmuş olur. Aramaya, EB-sonekli ağaç ile devam edilir. EB-sonekli ağaç aşağıdaki gibidir:

EB-sonekli ağaç

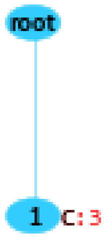


Şimdi de EB-sonekli ağaçtaki öğelerin destek değerlerini bulalım. Bu destek değerleri, C:2 ve A:1 şeklindedir. $AbsMinSupp = 2$ eşik değerini sadece C öğesi geçtiği için, buradan sadece CEB:2 sık öğeseti gelir. CEB-sonekli ağaca bakıldığında, aşağıdaki şekilden de görüleceği gibi, sadece kök düğümünden ibaretttir; dolayısıyla buradan daha başka sık öğeseti gelmez.

CEB-sonekli ağaç

EB-sonekli ağaç tamamlandıktan sonra, sıradaki AB-sonekli ağaca geçilebilir. Bu ağaç, aşağıdaki gibidir ve $AbsMinSupp = 2$ eşik değerini geçen sadece C:3 öğesi bulunmaktadır; dolayısıyla buradan sadece CAB:3 sık öğesi gelir. CAB-sonekli ağaç da sadece kök düğümünden ibaret olduğu için AB- ile biten sık öğesetlerinin tamamı bulunmuş olur.

AB-sonekli ağaç



CAB-sonekli ağaç

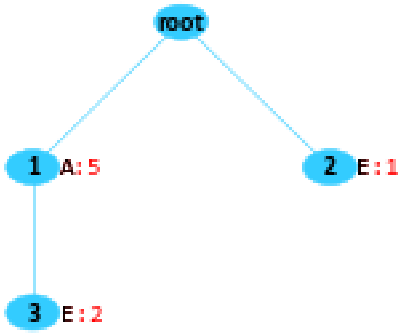
Şimdi sıra son olarak CB- ile biten sık öğesetlerine gelmiştir. Bunun için önce CB-sonekli ağacın bulunması gerekir. CB-sonekli ağaç sadece kök düğümünden ibaret olduğu için buradan başka sık öğeseti gelmez.

CB-sonekli ağaç

Böylece, B-sonekli sık öğesetlerinin tamamı bulunmuş olur. B-sonekli sık öğesetleri şu şekildedir. B:5, EB:3, CEB:2, CAB:3, AB:3, CB:4.

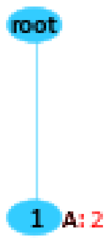
Şimdi, benzer şekilde, sıralamada bir sonraki öğe olan C öğesi ile biten sık öğesetlerini bulalım. İlk olarak C-sonekli ağaç aşağıdaki gibi elde edilir:

C-sonekli ağaç



C-sonekli ağaç elde edildikten sonra, bu ağaçta bulunan öğelerin destek değerlerinin sayılması işlemine geçilir. C-sonekli ağaçtaki öğelerin destek değerleri sırasıyla, A:5 ve E:3 şeklindedir. $AbsMinSupp = 2$ olduğu için, bu eşik değerini A ve E öğeleri geçer; dolayısıyla buradan AC:5, EC:3 şeklinde iki tane sık öğeseti bulunmuş olur. Aramaya, EC-sonekli ağaç ile devam edilir. EC-sonekli ağaç aşağıdaki gibidir:

EC-sonekli ağaç



Şimdi de EC-sonekli ağaçtaki öğelerin destek değerlerini bulalım. Bu destek değerleri sadece A:2 şeklindedir. $AbsMinSupp = 2$ eşik değerini A öğesi geçtiği için, buradan AEC:2 sık öğeseti gelir. AEC-sonekli ağaca bakıldığında, aşağıdaki şekilden de görüleceği gibi, sadece kök düğümünden ibarettir; dolayısıyla buradan daha başka sık öğeseti gelmez.

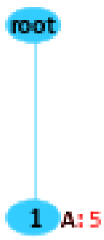
AEC-sonekli ağaç

Aramaya, AC-sonekli ağaç ile devam edilir. AC-sonekli ağaç aşağıdaki görüldüğü gibi, sadece kök düğümünden ibarettir; dolayısıyla buradan daha başka sık öğeseti gelmez.

AC-sonekli ağaç

Şimdi, benzer şekilde, sıralamada bir sonraki öğe olan E öğesi ile biten sık öğesetlerini bulalım. İlk olarak E-sonekli ağaç aşağıdaki gibi elde edilir:

E-sonekli ağaç



E-sonekli ağaçta sık öğe olarak sadece A:5 öğesi vardır, buradan sık öğeseti olarak AE:5 sık öğeseti gelir. AE-sonekli ağaca bakıldığında da onun sadece kök düğümünden ibaret olduğu görülür; dolayısıyla buradan daha başka sık öğeseti gelmez.

AE-sonekli ağaç

Böylece, E- ile biten sık öğesetleri şu şekilde bulunmuş olur: E:7 ve AE:5.

Son olarak A- ile biten sık öğesetlerinin bulunmasına gerek kalmaz çünkü, A-sonekli ağaç, aşağıdaki şekilden de görüleceği gibi, sadece kök düğümünden ibarettir; dolayısıyla buradan daha başka sık öğeseti gelmez.

A-sonekli ağaç

Böylece, ağaçtaki tüm sık öğesetleri bulunmuş olur. FPtree algoritmasının ilk aşamasında elde edilen bu ağacın bazı önemli özellikleri vardır:

-- Bütünlük: Sık öğesetlerini bulmak için gerekli tüm bilgiyi barındırır.

-- Sıkıştırılmışlık: Sık olmayan öğesetleri ağaçta bulunmaz.

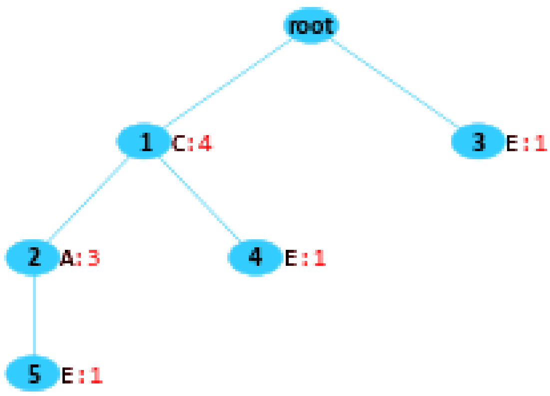
-- Destek sayısı daha büyük olan öğeler köke daha yakındır.

-- Hafızada orjinal veritabanından daha az yer kaplar.

FPtree algoritmasının gerçeklenmesi, ağacın oluşturulması ve ağaçtan sık öğesetlerinin elde edilmesi olmak üzere iki aşamadan oluşmaktadır. Burada en önemli konu, ağacın gösteriliminin nasıl yapılacağıdır, yani ağaçtaki düğümlere ilişkin öğesetleri, bunların destek değerleri ve bu düğümlerin birbirleriyle olan çocuk (child) ve ebeveyn (parent) ilişkilerinin nasıl ifade edileceğidir. Bilindiği kadarıyla, dizi (array), matris (matrix) ve nesne (object) olmak üzere üç farklı yaklaşım vardır. Kullanılan programlama dilinin durumuna göre, bu üç farklı yaklaşımın hızları farklılık gösterebilir.

Nesne yaklaşımında, ağaç nesnesi, düğüm nesnelerinden oluşmaktadır. Her bir düğüm nesnesi, o düğümdeki öğesetinin adı, öğesetinin destek değeri ve öğesetinin diğer düğümlerle olan ilişkileri gibi bilgileri barındırır. Bu şekildeki düğüm nesnelerinin bir araya gelmesiyle ağaç nesnesi oluşur.

Diğer taraftan, dizi (array) yaklaşımında ise, öğesetleri arasındaki ilişkilerin bulunduğu bir anadizi vardır. Bu anadizinin her bir elemanı bir öğesetini temsil eder. Bu anadizi ile aynı boyutta başka bir dizi bu öğesetinin adı, başka bir dizide de bu öğesetinin destek değeri bulunur. Anadizide, öğesetleri arasındaki ilişkileri ifade etmek için diziye uygun tamsayılar atanır. Örneğin, aşağıdaki ağaç dizi yaklaşımı ile şu şekilde ifade edilir:



$$
\begin{aligned}
\text{anadizi} &= [-1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 2] \\
\text{öğeseti} &= [-1 \quad \mathrm{C} \quad \mathrm{A} \quad \mathrm{E} \quad \mathrm{E} \quad \mathrm{E}] \\
\text{sayı} &= [-1 \quad 4 \quad 3 \quad 1 \quad 1 \quad 1]
\end{aligned}
$$

Boş bir anadizi ile başlandığında, anadizi'deki ilk eleman olan kök düğümü temsil etmek için her zaman -1 kullanılır. 1 nolu düğümdeki C öğesini temsil etmek için onun ebeveyni olan kök düğümünün anadizideki indisi (0), anadizinin bir sonraki elamanına yazılır. Öğeseti dizisine C öğesi ve sayı dizisine de C öğesinin sayısı olan 4 eklenir. Böylece anadizi $[-1 \quad 0]$, öğeseti dizisi $[-1 \quad \mathrm{C}]$ ve sayı dizisi de $[-1 \quad 4]$ haline gelir. Sırada, örneğin 2 nolu düğümdeki A öğesi varsa, onun ebeveyni olan C öğesi anadizide 1 nolu indekste yer aldığı için anadiziye 1 elemanı eklenir, öğeseti ve sayı dizilerine de sırasıyla A ve 3 eklenir. Böylece anadizi $[-1 \quad 0 \quad 1]$, öğeseti dizisi $[-1 \quad \mathrm{C} \quad \mathrm{A}]$ ve sayı dizisi de $[-1 \quad 4 \quad 3]$ haline gelir. 3 nolu düğümdeki E öğesi ile devam etmek istersek, anadiziye 0 elemanını eklememiz gerekir çünkü onun ebeveyninin anadizideki indeksi 0'dır. Böylece anadizi $[-1 \quad 0 \quad 1 \quad 0]$, öğeseti dizisi $[-1 \quad \mathrm{C} \quad \mathrm{A} \quad \mathrm{E}]$ ve sayı dizisi de $[-1 \quad 4 \quad 3 \quad 1]$ haline gelir. 4 nolu düğümdeki E öğesinin ebeveyni olan C öğesi anadizide 1 nolu indekste yer aldığı için anadiziye 1 elemanı eklenir, öğeseti ve sayı dizilerine de sırasıyla E ve 1 eklenir. Son olarak, 5 nolu düğümdeki E öğesinin ebeveyni olan A öğesi anadizide 2 nolu indekste yer aldığı için anadiziye 2 elemanı eklenir, öğeseti ve sayı dizilerine de sırasıyla E ve 1 eklenirse, gösterilim

$$\begin{aligned} \text{anadizi} &= [-1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 2] \\ \text{öğeseti} &= [-1 \quad \mathrm{C} \quad \mathrm{A} \quad \mathrm{E} \quad \mathrm{E} \quad \mathrm{E}] \\ \text{sayı} &= [-1 \quad 4 \quad 3 \quad 1 \quad 1 \quad 1] \end{aligned}$$

haline gelir. Böylelikle, anadizi, öğeseti dizisi ve sayı dizilerine bakılarak ağaçta hangi düğümde hangi öğenin bulunduğu, bu öğesinin sayısı, ebeveyni ve çocukları belirlenebilir.

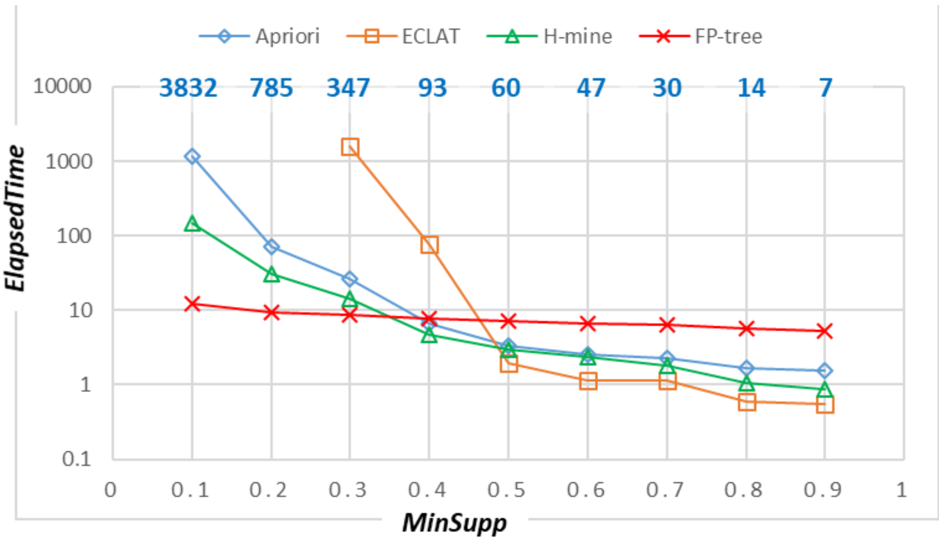
Ağacın temsil edilmesinde kullanılan diğer bir yaklaşım ise matris yaklaşımıdır. Bu yaklaşım, dizi yaklaşımına benzemektedir. Matris yaklaşımının ayrıntılarına burada değinilmeyecektir.

3.6 Karşılaştırma Tabloları

Bu kısımda, bir Python uygulaması yapılacaktır. Seçilen bir veritabanı için *MinSupp* eşik değeri 0.1 ile 0.9 arasında 0.1 aralıklarla değiştirilerek algoritmaların sık öğesetleri döndürme süreleri ölçülecektirr. Not edilmelidir ki aynı veritabanı ve *MinSupp* eşik değeri için algoritmaların döndürdükleri sık öğesetleri aynıdır. Aşağıdaki örnek tablo ve grafikte olduğu gibi bir karşılaştırma yapılacaktır.

MinSupp	**Apriori**	**ECLAT**	**H-mine**	**FP-tree**	**#of FIs**
0.1	1154.02		146.12	12.37	**3832**
0.2	69.93		30.39	9.26	**785**
0.3	26.75	1587	14.41	8.71	**347**
0.4	6.65	75.65	4.57	7.70	**93**
0.5	3.31	1.93	2.93	7.24	**60**
0.6	2.56	1.12	2.37	6.54	**47**
0.7	2.25	1.10	1.81	6.46	**30**
0.8	1.68	0.58	1.04	5.72	**14**
0.9	1.57	0.54	0.87	5.16	**7**

Aynı sonuçlar, aşağıdaki şekildeki gibi grafik olarak da verilecektir.



4) SIK ÖĞESETLERİNDEN KURAL ÇIKARIMI

4.1 İlişkisel Kurallar

FIM algoritmaları ile bulunan sık öğesetleri arasındaki ilişkileri bulmak için İlişkisel Kural Madenciliği (Association Rule Mining - ARM) yöntemleri kullanılmaktadır. Bu kurallar sayesinde, örüntüler arasındaki ilişkileri belli sayısal değerlerle ifade etmek mümkün olmaktadır. Örnek olarak, aşağıdaki gibi verilen bir veritabanını ele alalım:

TID	İşlem (Transaction)
T_1	$\{a, c, d\}$
T_2	$\{b, c, e\}$
T_3	$\{a, b, c, e\}$
T_4	$\{b, e\}$
T_5	$\{a, b, c, e\}$

Bu veritabanı ve $MinSupp = 0.4$ için sık öğesetleri bir FIM algoritmasıyla aşağıdaki şekilde bulunmuş olsun:

	Öğeseti	**RelSupp**		**Öğeseti**	**RelSupp**
#1	$\{a\}$	0.6	#8	$\{b, c\}$	0.6
#2	$\{b\}$	0.8	#9	$\{b, e\}$	0.8
#3	$\{c\}$	0.8	#10	$\{c, e\}$	0.6
#4	$\{e\}$	0.8	#11	$\{a, b, c\}$	0.4
#5	$\{a, b\}$	0.4	#12	$\{a, b, e\}$	0.4
#6	$\{a, c\}$	0.6	#13	$\{a, c, e\}$	0.4
#7	$\{a, e\}$	0.4	#14	$\{b, c, e\}$	0.6
			#15	$\{a, b, c, e\}$	0.4

Bu şekilde bulunan sık öğesetleri arasındaki ilişkiyi ifade etmek için şu notasyon kullanılacaktır: Verilen bir veritabanı ve $MinSupp$ değeri için, X ve Y, destek değerleri sıfır olmayan, yani $Supp(X) \neq 0$ ve $Supp(Y) \neq 0$, ve kesişim kümesi boş-küme ($X \cap Y = \emptyset$) olan herhangi iki **sık** öğeseti olmak üzere bu iki öğeseti arasındaki ilişkisel kural (rule veya implication) $X \rightarrow Y$ notasyonu ile gösterilir ve "X öğesetinin olduğu bir işlemde Y öğeseti de vardır" anlamına gelir. Bu notasyonda X *öncül* (*premise*), Y ise ardıl (*consequent*) olarak adlandırılmaktadır.

Örneğin, yukarıda bulunan sık öğesetleri için $\{a, e\} \rightarrow \{c\}$ gibi bir kural, "$\{a, e\}$ öğesetinin olduğu bir işlemde $\{c\}$ öğeseti de vardır" anlamına gelir.

4.2 Güven (Confidence)

Ancak böyle bir kural için akla şu sorular gelmektedir: $\{a, e\}$ öğesetinin olduğu bir işlemde $\{c\}$ öğesetinin de bulunması ne kadar olasıdır? Biz bu kurala ne kadar güvenebiliriz? Burada, kuralın güvenirliğini ifade etmek için olasılıksal bir değer vermek gerekir ki bu da bizi kural ile ilgili olarak "güven (confidence)" kavramına götürür.

$X \rightarrow Y$ şeklindeki bir kuralın güvenirliğini ifade etmek için, literatürde "güven (confidence)" kavramı önerilmiştir. $X \rightarrow Y$ şeklindeki bir kuralın *güven* değeri $Conf(X \rightarrow Y)$ ile gösterilir ve şu şekilde hesaplanır:

$$Conf(X \rightarrow Y) = \frac{Supp(X \cup Y)}{Supp(X)}.$$

Güven değerinin hesabında, destek değeri $Supp(.)$ Olarak bağıl destek $RelSupp$ kullanılabileceği gibi mutlak destek $AbsSupp$ değeri de kullanılabilir, aynı sonucu üretirler. $Conf(X \rightarrow Y)$ güven değeri, X ve Y öğesetlerinin birleşiminden (union) oluşan $X \cup Y$ öğesetinin destek değerinin, X öğesetinin destek değerine oranı ile hesaplanır. Downward-closure özelliğinden hatırlanacağı gibi, her zaman $Supp(X \cup Y) \leq Supp(X)$ olmaktadır. Dolayısıyla, güven değeri her zaman $[0 \quad 1]$ aralıpında olacaktır, yani

$$0 \leq Conf(X \rightarrow Y) \leq 1.$$

Bilindiği gibi, bir X öğesetinin sık öğeseti olabilmesi için $MinSupp \leq Supp(X)$ şartının sağlanması gerekir. Benzer şekilde, $Supp(X) = 0$ ve $Supp(Y) = 0$ olmak üzere, $X \rightarrow Y$ gibi bir ilişkinin bir *ilişkisel kural* (*association rule*) olabilmesi için şu üç koşulu aynı anda sağlaması gerekir:

(1) $X \cap Y = \emptyset$
(2) $MinSupp \leq Supp(X \cup Y)$
(3) $MinConf \leq Conf(X \rightarrow Y)$

Burada, $MinConf$ değeri, kuralın sahip olması gereken en küçük güven değeridir. Bu güven değerinin altında bir güven değerine sahip bir ilişki, ilişkisel kural olmaz. Böylece, işlemlerden (transactions) oluşan bir veritabanından ilişkisel kurallar çıkarabilmek için hem minimum destek değeri $MinSupp$, hem de minimum güven değeri $MinConf$ verilmesi gerekir ki buna literatürde *destek-güven çerçevesi* (*support-confidence framework*) denmektedir.

Örneğe devam edilirse, $MinSupp = 0.40$ değeri için bulunan aşağıdaki sık öğesetlerinin ele alalım:

	Öğeseti	RelSupp		Öğeseti	RelSupp
#1	$\{a\}$	0.6	#8	$\{b, c\}$	0.6
#2	$\{b\}$	0.8	#9	$\{b, e\}$	0.8
#3	$\{c\}$	0.8	#10	$\{c, e\}$	0.6
#4	$\{e\}$	0.8	#11	$\{a, b, c\}$	0.4
#5	$\{a, b\}$	0.4	#12	$\{a, b, e\}$	0.4
#6	$\{a, c\}$	0.6	#13	$\{a, c, e\}$	0.4
#7	$\{a, e\}$	0.4	#14	$\{b, c, e\}$	0.6
			#15	$\{a, b, c, e\}$	0.4

Bu sık öğesetleri arasındaki ilişkisel kuralları bulmak için, minimum güven değeri $MinConf = 0.50$ olarak belirlenirse, çok sayıda ilişkisel kural bulunabilir. Bunlardan bir tanesi de $\{a, c\} \rightarrow \{b, e\}$ şeklindedir. Bu kuralın öncülünün destek değeri $Supp(\{a, c\}) = 0.60$, ardılının destek değeri $Supp(\{b, e\}) = 0.80$ şeklinde olup bunlarin birleşimi olan öğesetinin destek değeri $Supp(\{a, b, c, e\}) = 0.40$ şeklinde olup kuralın güven değeri şu şekildedir:

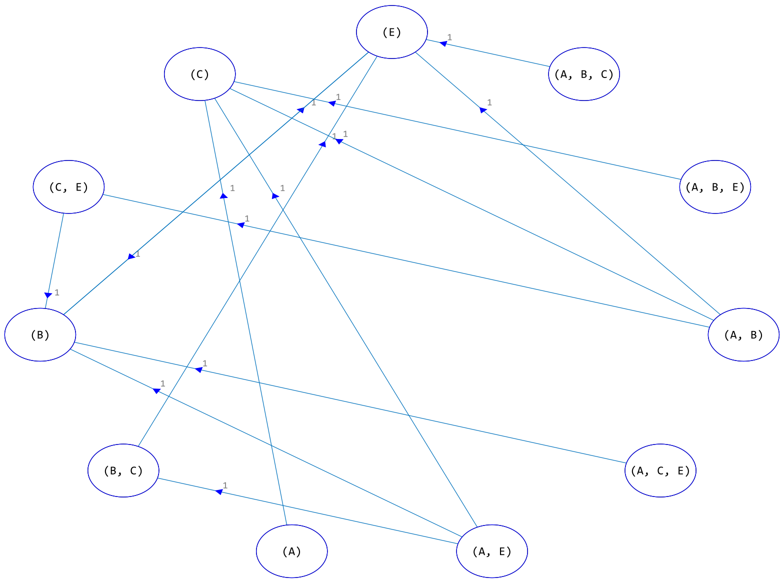
$$Conf(\{a, c\} \rightarrow \{b, e\}) = \frac{Supp(\{a, b, c, e\})}{Supp(\{a, c\})} = \frac{0.4}{0.6} = 0.66.$$

Bu kuralın anlamı şudur: "$\{a, c\}$ öğesetinin olduğu bir işlmde, $\{b, e\}$ öğesetinin de bulunma olasılığı 0.66'dır." Görüldüğü gibi, güven değeri, kuralın hangi olasılıkla geçerli olduğunu göstermektedir; dolayısıyla, uygulamalarda olabildiğince yüksek seçilir. Bu örnekte, güven değeri $MinConf = 0.95$ olarak seçilirse aşağıdaki kurallar elde edilir:

$MinSupp = 0.40$
$MinConf = 0.95$

premise → consequent	confidence
$\{e\} \rightarrow \{b\}$	1.00
$\{b\} \rightarrow \{e\}$	1.00
$\{a\} \rightarrow \{c\}$	1.00
$\{c, b\} \rightarrow \{e\}$	1.00
$\{e, c\} \rightarrow \{b\}$	1.00
$\{a, b\} \rightarrow \{e,c\}$	1.00
$\{a, e\} \rightarrow \{c,b\}$	1.00
$\{a, e\} \rightarrow \{c\}$	1.00
$\{a, c, b\} \rightarrow \{e\}$	1.00
$\{a, e, b\} \rightarrow \{c\}$	1.00
$\{a, e, c\} \rightarrow \{b\}$	1.00
$\{a,b\} \rightarrow \{e\}$	1.00
$\{a, e\} \rightarrow \{b\}$	1.00
$\{a, b\} \rightarrow \{c\}$	1.00

Bu kuralları aşağıdaki şekildeki gibi görsel olarak da görmek mümkündür. Burada, dairelerin içindekiler örüntülerdir (sık öğesetleri), daireleri birleştiren okların yönü kuralın yönünü, okun yanındaki sayı ise kuralın güven (confidence) değerini göstermektedir.



Eğer $MinSupp = 0.80$ ve $MinConf = 0.95$ seçilirse, sadece iki kural elde edilir.

$MinSupp = 0.80$
$MinConf = 0.95$

premise → consequent	confidence
$\{e\} \rightarrow \{b\}$	1.00
$\{b\} \rightarrow \{e\}$	1.00

Görüldüğü gibi bazı veritabanları için confidence değerlerinin tamamı 1.00 çıkabilmektedir. Dolayısıyla bu durum, daha farklı ilave ölçütler bulmaya yöneltmektedir.

4.3 İlgi (Interest)

İlgi (interest) kavramı, olasılık teorisindeki bağımsızlık kavramına benzeyen bir kavramdır. Hatırlanacağı gibi, $\mathbb{A}$ ve $\mathbb{B}$ iki farklı olay, $\mathbb{AB}$ bu iki olayın aynı anda oluştuğu kesişim olayı ve $P(\mathbb{A})$, $P(\mathbb{B})$ ve $P(\mathbb{AB})$ sırasıyla bunların olasılıkları olmak üzere, eğer

$$P(\mathbb{AB})=P(\mathbb{A})P(\mathbb{B})$$

ise o zaman $\mathbb{A}$ ve $\mathbb{B}$ olayları bağımsızdır. Veritabanındaki öğesetlerini olay, bunların bağıl desteklerini de olasılık olarak ele aldığımızda, olayların bağımsızlığı kavramına benzer olarak iki öğesetinin ilgisi (interest) kavramı karşımıza çıkar. X ve Y gibi iki öğesetinin destek değerleri $Supp(X)$ ve $Supp(Y)$ ve bunların birleşim öğeseti $X \cup Y$ öğesetinin destek değeri $Supp(X \cup Y)$ olmak üzere, $X \rightarrow Y$ şeklindeki ilişkisel bir kuralın ilgi (interest) değeri

$$Inte(X \rightarrow Y) = |Supp(X \cup Y) - Supp(X)Supp(Y)|$$

şeklinde verilmektedir. $Inte(X \rightarrow Y) = 0$ olması durumunda X ve Y öğesetleri bağımsızdır ve buradan ilişkisel bir kural türetilemez. Yukarıdaki örneğe geri dönülecek olursa, ilgi değerleri de eklendiğinde sonuç tablosu şu hale gelir:

$MinSupp = 0.40$
$MinConf = 0.95$
$MinInte = 0.08$

premise → consequent	**confidence**	**interest**
$\{e\} \rightarrow \{b\}$	1.00	0.16
$\{b\} \rightarrow \{e\}$	1.00	0.16
$\{a\} \rightarrow \{c\}$	1.00	0.12
$\{c, b\} \rightarrow \{e\}$	1.00	0.12
$\{e, c\} \rightarrow \{b\}$	1.00	0.12
$\{a, b\} \rightarrow \{e, c\}$	1.00	0.16
$\{a, e\} \rightarrow \{c, b\}$	1.00	0.16
$\{a, e\} \rightarrow \{c\}$	1.00	0.08
$\{a, c, b\} \rightarrow \{e\}$	1.00	0.08
$\{a, e, b\} \rightarrow \{c\}$	1.00	0.08
$\{a, e, c\} \rightarrow \{b\}$	1.00	0.08
$\{a, b\} \rightarrow \{e\}$	1.00	0.08
$\{a, e\} \rightarrow \{b\}$	1.00	0.08
$\{a, b\} \rightarrow \{c\}$	1.00	0.08

4.4 Örnek Sonuçlar

Bu kısımda, örnek bir veritabanı için bir Python uygulaması yapılacaktır.

REFERANSLAR

[1] Agrawal R, Imielinski T and Swami A, "Mining Association Rules between Sets of Items in Large Databases," Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 207–216, 1993.

[2] Zaki MJ, "Scalable Algorithms for Association Mining," IEEE Transactıons on Knowledge and Data Engineerıng, Vol. 12, No. 3, pp. 372–390, 2000.

[3] Pei J, Han J, Lu H, Nishio S, Tang S and Yang D, "H-Mine: Fast and space-preserving frequent pattern mining in large databases," IIE Transactions, Vol. 39, pp. 593–605, 2007.

[4] Han J, Pei J and Yin Y, "Mining Frequent Patterns without Candidate Generation," Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000.

[5] http://www.philippe-fournier-viger.com/spmf/index.php