**GRAFİKLER** **7**

Bilgisayar Bilimi ve Matematikteki birçok problem, bir dizi durum ve bu durumlar arasındaki geçişlerin bir kümesine indirgenebilir. Bir grafik, bu tür problemlerin matematiksel bir temsilidir. Son bölümde, trees (ağaçların) Bilgisayar Bilimi'nde çeşitli amaçlar için hizmet ettiğini gördük. Trees ( Ağaçlar) grafiktir. Ancak, grafikler trees (ağaçlardan) daha genel bir kavramdır. Bir problemin detaylarını soyutlayarak en basit haliyle incelemek, çoğu zaman yeni bir anlayışa yol açar. Sonuç olarak, birçok algoritma grafik teorisi araştırmalarından ortaya çıkmıştır. Grafik teorisi ilk olarak matematikçiler tarafından incelenmiştir. Grafik teorisindeki birçok algoritma, onları geliştiren veya keşfeden matematikçinin adıyla anılmaktadır. Dijkstra ve Kruskal, bu tür matematikçilerden ikisidir ve bu bölüm, onların geliştirdiği algoritmaları ele almaktadır.

Bir grafiği temsil etmek, birkaç farklı şekilde yapılabilir. Bir grafiği temsil etmenin doğru yolu, uygulanacak olan algoritmaya bağlıdır. Grafik teorisi problemleri, grafikteki iki durum veya düğüm arasında bir yol bulma, bir grafikteki en kısa yolu bulma gibi birçok sorunu içerir. Grafikleri incelemekten ortaya çıkan birçok algoritma vardır. Bu problemlerin formülasyonunu anlayabilmek için, bu bölümde sunulan bir miktar grafik notasyonu öğrenmek faydalıdır.

**7.1 Bölüm Hedefleri**

Bu bölüm, grafiklerin temsilini ele alır. Ayrıca birkaç grafik algoritmasını da kapsar. Bir grafiğin derinlik öncelikli araması ve genişlik öncelikli araması sunulmuştur. Dijkstra algoritması, Bilgisayar Bilimi'nde ünlüdür ve ağ kurulumundan inşaat planlamasına kadar birçok uygulaması vardır. Kruskal algoritması, minimum ağırlıklı genişleyen tree bulmak için kullanılan bir diğer ünlü algoritmadır. Bölümün sonunda, grafik teorisinin temel bir anlayışına sahip olmalı ve Bilgisayar Bilimi'ndeki birçok problemin grafikler şeklinde nasıl ifade edilebileceğini anlamalısınız.

Başlangıç olarak, bazı notasyonları ve bir grafiğin derinlik öncelikli aramasını inceleyeceğiz. Ardından, grafiklerle ilgili bazı ilginç soruları yanıtlayan birkaç Greedy Algoritmayı inceleyeceğiz. Greedy algoritmalar, çözüm bulurken asla yanlış bir seçim yapmayan algoritmalardır.

Bu algoritmalardan ikisini, Kruskal’ın Algoritması ve Dijkstra’nın Algoritması’nı inceleyeceğiz; her ikisi de ilgili problemlerini çözmek için algoritmayı formüle eden kişilerin adlarıyla anılmaktadır.

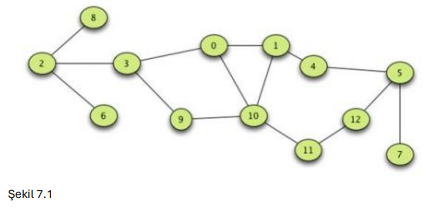
**7.2 Grafik Gösterimi**

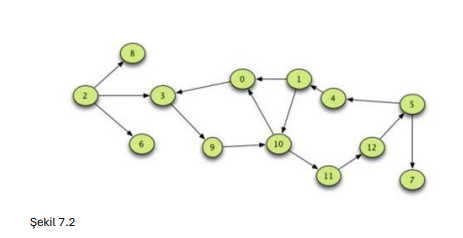
Bu bölümdeki grafik tanımlarında küçük bir notasyon yardımcı olacaktır. Bir küme, öğelerin sırasız bir koleksiyonudur. Örneğin, V = {0,1,2,3,4,5,6,7,8,9,10,11,12} ilk 13 doğal sayının kümesidir. Bir kümenin alt kümesi, onun superset (süperkümesinden) öğelerin bir koleksiyonudur, bu koleksiyon boş da olabilir. U = {5, 8, 2} kümesi, V kümesinin bir alt kümesidir. Bir kümenin kardinalitesi, onun büyüklüğü veya eleman sayısıdır. V kümesinin kardinalitesi |V| olarak yazılır. V kümesinin kardinalitesi 13, U'nun kardinalitesi ise 3'tür, yani |V| = 13 ve |U| = 3.

Bir G = (V,E) grafiği, V adında bir dizi köşe (veya düğüm) ve E adında bir dizi kenar ile tanımlanır. Kenar kümesi, her bir öğesi 2 kardinaliteye sahip olan V'nin alt kümeleridir. Başka bir deyişle, kenarlar, köşe çiftleriyle gösterilir. Şekil **7.1**'de verilen basit, yönsüz grafiği düşünün. V = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} ve E = {{0, 1}, {0, 3}, {0, 10}, {1, 10}, {1, 4}, {2, 3}, {2, 8}, {2, 6}, {3, 9}, {5, 4}, {5, 12}, {5, 7}, {11, 12}, {11, 10}, {9, 10}} bu grafiği tanımlar. Her kenar kendisi bir kardinalitesi 2 olan bir küme olduğundan, her kenar kümesindeki köşelerin sırası önemli değildir. Örneğin, {1, 4} ve {4, 1} aynı kenardır.

Pek çok problem, bir grafik terimiyle formüle edilebilir. Örneğin, iki ülke arasındaki sınırları paylaştıklarında aynı renkte olmamaları için bir haritayı boyamak için kaç renk gerektiğini sorabiliriz. Bu problemde, Şekil **7.1**'deki köşeler ülkeleri temsil eder ve sınır paylaşan iki ülke arasında bir kenar bulunur. Problem, o zaman her bir köşeyi öyle bir şekilde renklendirmek için gereken minimum renk sayısını bulmak olarak yeniden ifade edilebilir, böylece kenar paylaşan hiçbir iki köşe aynı renge sahip olmasın.

Yönlendirilmiş bir grafik G = (V, E), yönsüz bir grafiğin aynı şekilde tanımlanır, ancak kenar kümesi E, alt kümeler yerine çiftlerden oluşan bir küme olarak tanımlanır. E = {(vi, vj) where vi, vj ∈ V } şeklinde tanımlanması, kenarların yalnızca bir yönde geçilebileceği anlamına gelir. Şekil **7.2**'de, (10, 0) kenarı boyunca, 10. köşesinden 0. köşesine hareket edebiliriz, ancak 0'dan 10'a hareket edemeyiz, en azından başka köşelerden geçmeden çünkü (0, 10) kenarı E kümesinde yer almamaktadır.





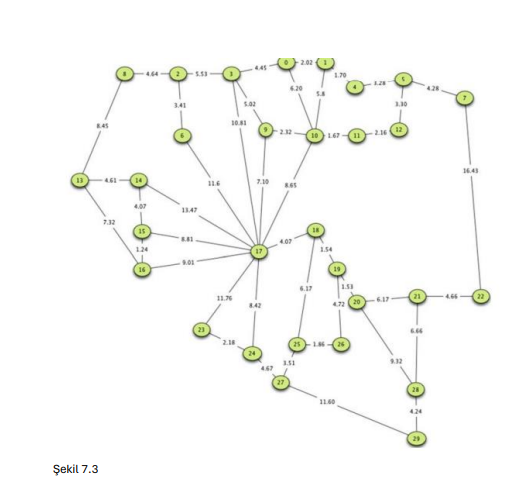
Bir grafikte bir yol, tekrar edilmeyen bir dizi kenardan oluşur ve bir köşeden başka bir köşeye gitmek için geçilebilir. Bir grafikteki bir döngü, aynı köşe ile başlayıp aynı köşe ile biten bir yoldur. Son bölümde, bilgisayar biliminde trees ele alınmıştı. Şimdi, grafik teorisinden bazı notasyonlarla donanmış olarak, bir tree resmi tanımını verebiliriz. Bir tree, yönlendirilmiş, bağlı ve döngüsüz bir grafiktir. Döngüsüz bir grafik, içinde herhangi bir döngü olmayan bir grafiktir.

Bazen grafik teorisinde, tree, yönlendirilmiş bir grafik olma gerekliliği kaldırılarak döngüsüz bağlı bir grafik olarak tanımlanır. Bu durumda, bir tree, tamamen bağlı olan ancak herhangi iki köşe arasında yalnızca bir yol bulunan bir grafik olarak tanımlanabilir

Yönlendirilmiş ve yönsüz grafikler, birçok farklı türde problemi modellemek için kullanılabilir. Şekil **7.1**'deki grafik, bir CPU'daki register tahsisini temsil edebilir. Köşeler, sembolik olarak adlandırılmış register'ları temsil edebilir ve aynı anda kullanılan iki register arasında bir kenar bulunur. Sorulabilecek soru şu olabilir: 'Bu hesaplamanın sembolik register'ları için makinenin kaç fiziksel register'ına ihtiyaç vardır?'

Meğerse register tahsisi ve harita boyama aynı problemi temsil ediyormuş. Detayları soyutladığımızda, problem bir grafik boyama problemine indirgenir. 'Haritayı boyamak için kaç renk gereklidir?' sorusunun cevabı, 'Bu hesaplama için kaç fiziksel register gereklidir?' sorusuna ve tam tersine cevap verir

Ağırlıklı bir grafik, her kenarına bir ağırlık atanmış bir grafiktir. Daha resmi bir şekilde, ağırlıklı bir G = (V, E, w) grafiği, verilen köşe kümesi V ve kenar kümesi E ile tanımlanmış bir grafiktir. Ayrıca, bir ağırlıklı grafiğin, kenarları gerçel sayılara haritalayan bir ağırlık fonksiyonu w'yu vardır. Yani w fonksiyonunun imzası w: E → Gerçek olarak verilir. Ağırlıklı grafikler, birçok farklı problemin durumunu temsil etmek için kullanılabilir. Örneğin, bir ağırlıklı grafik, yollar ve kavşaklarla ilgili bilgi verebilir. Maliyet/fayda analizi bazen ağırlıklı grafikler terimleriyle ifade edilebilir. Ağırlıklar, bir ağdaki düğümler arasındaki ağ bağlantılarının mevcut kapasitesini temsil edebilir.

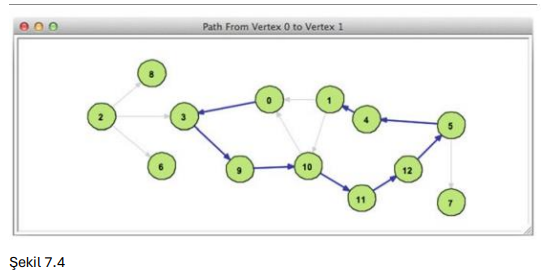


Ağırlıklı bir grafik, birçok farklı türdeki problemin durumunu temsil etmek için kullanılabilir. Şekil **7.3**, yolları ve kavşakları temsil eden ağırlıklı bir grafiği göstermektedir.

**7.3 Grafik Arama**

Pek çok problem grafik teorisi terimleriyle formüle edilmiştir. Daha yaygın problemlerden biri, bir grafikte bir köşeden diğerine giden bir yol keşfetmektir. Sorulabilecek soru şu olabilir: vi köşesinden vj köşesine bir yol var mı ve varsa, oraya ulaşmak için hangi kenarlardan geçmelisiniz? Bir grafikte derinlik öncelikli arama yapmak, Bölüm **6**'da ilk kez sunulan algoritmaya benzer, ancak grafikte bir döngüye takılmamaya dikkat etmeliyiz.

Şekil **7.2**'deki yönlendirilmiş grafikte 0. köşesinden 1. köşesine bir yol aramayı düşünün. Şekil **7.4**'teki mavi çizgiler, bu köşeler arasındaki yolu vurgulamaktadır. Grafikte, çoğu durumda yalnızca bir seçim olduğu görülüyor. Ancak, arama 10. köşeye ulaştığında iki kenar arasında bir seçim yapmamız gerekiyor. Bir kenar bizi daha önce ziyaret ettiğimiz 0. köşesine geri götürüyor. Diğer kenar ise bizi nihai yola daha da yaklaştırıyor. Başka bir seçim, 5. köşede yapılır. Eğer 7'ye giden kenar yanlış bir şekilde incelenirse, geri dönüp 4. köşesine giden diğer kenarı denemek için bir yolumuz olmalı.



Bu şekilde bir grafikte arama yapmak, Bölüm **6**'da ilk kez tartışıldığı gibi depth first search(derinlik öncelikli arama) olarak da adlandırılır ve geri izleme yeteneği gerektirir. 5. köşe ile karşılaşıldığında, 7. köşeye gitmeye karar verilirse, bu tercihi düzeltmek ve bunun yerine 4. köşeye gitmek için geri adım atabilmemiz gerekir. Geri izleme, bir yığın veri yapısı veya özyineleme (recursion) ile yapılır.

Derinlik öncelikli arama, aynı zamanda grafikteki olası döngülerden kaçınmalıdır. Döngülerden kaçınmak, visited(ziyaret) edilen köşelerin bir kümesini tutarak yapılır. Bir köşe visited edildiğinde, visited edilen kümesine eklenir. Eğer bir köşe visited edilen kümede yer alıyorsa, o köşe daha sonra arama sırasında tekrar incelenmez, çünkü bir döngü aramayı aynı köşeye geri götürebilir.

Yinelemeli (yani özyinelemeli olmayan) bir grafik derinlik öncelikli arama algoritması, visited edilen kümesini boş bir küme olarak başlatmak ve geri izleme için bir yığın oluşturmakla başlar. Başlangıç köşesi, algoritmaya başlamak için yığına eklenir. Hedefi bulmak için, Bölüm **7.3.1**'de yapılan adımlara benzer adımlar yürütülür. Bu kod, pseudo-code ancak gerekli detayları sunar.

**7.3.1 Bir Grafiğin Yinelemeli İlk Derinlik Araması**

1 **def** graphDFS ( G, start, goal ) :

2 # G = ( V , E ) is the graph with vertices, V, and edges, E.

3 V,E = G

4 stack = Stack( )

5 visited = Set( )

6 stack.push(start)

7

8  **while not** stack.isEmpty( ) :

9 # A vertex is popped from the stack. This is called the current vertex.

10 current = stack.pop( )

11 # The current vertex is added to the visited set.

12 visited.add(current)

13

14 # If the current vertex is the goal vertex, then we discontinue the

15 # search reporting that we found the goal.

16  **if** current == goal:

17 **return** True # or return path to goal perhaps

18

19 # Otherwise, for every adjacent vertex, v, to the current vertex

20 #in the graph, v is pushed on the stack of vertices yet to search

21 # unless v is already in the visited set in which case the edge

22 # leading to v is ignored.

23 **for** v **in** adjacent(current , E) :

24  **if not** v **in** visited:

25 stack.push(v)

26

27 # If we get this far, then we did not find the goal.

28 **return** False # or return an empty path

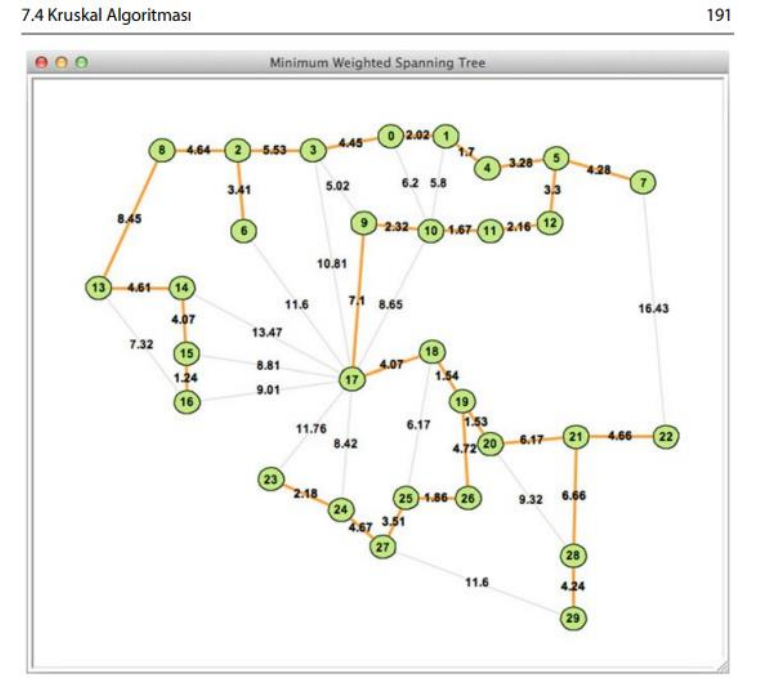
Eğer bölüm **7.3.1**'deki while döngüsü sona ererse, yığın boştu ve bu nedenle hedefe giden bir yol yok demektir. Bu algoritma, bir grafın derinlik öncelikli aramasını (depth first search) uygular. Ayrıca, yığına eleman eklemeyi depth first search’a yapılan bir özyinelemeli çağrı ile değiştirerek özyinelemeli olarak da uygulanabilir. Özyinelemeli olarak uygulandığında, depth first search fonksiyonu mevcut düğümü ve değiştirilebilir bir visited edilen kümesini alır veya hedefe giden yolu döndürür ya da alternatif olarak hedefin veya amacın bulunduğunu belirten bir boolean değeri döndürür. Şekil **7.4**’teki graf verildiğinde, arama True döndürmüştür.

Depth first search‘ın iteratif versiyonu, yığını bir kuyrukla değiştirerek bir graf üzerinde breadth first search (genişlik öncelikli arama) yapmak için değiştirilebilir. Genişlik öncelikli arama, tüm yolları aynı anda inceleyen kapsamlı bir aramadır, ancak aynı zamanda bir graf üzerinde herhangi iki düğüm arasındaki en kısa yolu, en az sayıda kenar ile bulacaktır. Genişlik öncelikli arama, büyük graflar üzerinde uygulandığında, pratikte kullanılacak kadar uzun sürebilir.

**7.4 Kruskal Algoritması**

Bir an için, kışın yolları kardan temizlemekle sorumlu olan ve beklenmedik bir kar miktarı nedeniyle parasız kalan bir ilçeyi düşünün. İlçe denetçisine, kışın geri kalan kısmında yalnızca gerekli yolları temizleyerek maliyetleri azaltması söylenmiştir. Denetçi, ilçedeki herhangi bir noktadan diğer bir noktaya seyahat edilebilmesi için kazınması gereken toplam en kısa mil sayısını bulmak istemektedir, ancak bu, mutlaka en kısa yol olmak zorunda değildir. İlçe denetçisi, temizlenen yol mil miktarını minimize etmek, ancak yine de ilçede ihtiyacınız olan her yere ulaşabileceğinizden emin olmak istemektedir.

Joseph Kruskal, 1928'den 2010'a kadar yaşamış bir Amerikalı bilgisayar bilimcisi ve matematikçiydi. Bu problemi hayal etti, ağırlıklı bir graf üzerinden bunu formüle etti ve bu problemi çözmek için bir algoritma geliştirdi. Algoritması, ilk kez Amerikan Matematiksel Derneği'nin Bildirileri'nde [5] yayımlandı ve genellikle Kruskal’ın Algoritması olarak adlandırılmaktadır.

Son bölümde, ikili arama gibi çeşitli algoritmalarda trees kullanarak tanıttık. Tanım, son bölümden değişmez. Ancak, grafik teorisi bağlamında trees, tüm olası graf kümelerinin bir alt kümesidir. Bir trees, içinde hiçbir döngü bulunmayan bir grafiktir. Ayrıca, bir tree, düğüm sayısından bir eksik kenar içermesi gerektiğini kanıtlamak oldukça kolaydır. Aksi takdirde, o bir tree olmazdı.

Açıkça, Şekil **7.3**'teki grafik bir tree değildir. Grafikte birçok döngü bulunmaktadır. Kruskal’ın makalesi, böyle bir grafik için minimum ağırlıklı yayılma tree bulmak için bir algoritma sunmuştur.Şekil **7.5**, Şekil **7.3**’teki grafik için bir minimum ağırlıklı yayılma tree ‘sini içerir ve tree kenarları turuncu ile vurgulanmıştır. "Minimum ağırlıklı yayılma tree" demiyoruz çünkü genel olarak birden fazla minimum ağırlıklı yayılma tree olabilir. Bu durumda, muhtemelen yalnızca bir tane mümkündür.

Kruskal’ın algoritması, açgözlü (greedy) bir algoritmadır. "greedy" terimi, algoritmanın her zaman alternatifler listesi sunulduğunda ilk alternatifi seçtiği ve seçim yaparken asla hata yapmadığı veya yanlış bir tercih yapmadığı anlamına gelir. Diğer bir deyişle, Kruskal’ın algoritmasında geri izleme (backtracking) gerekmez.

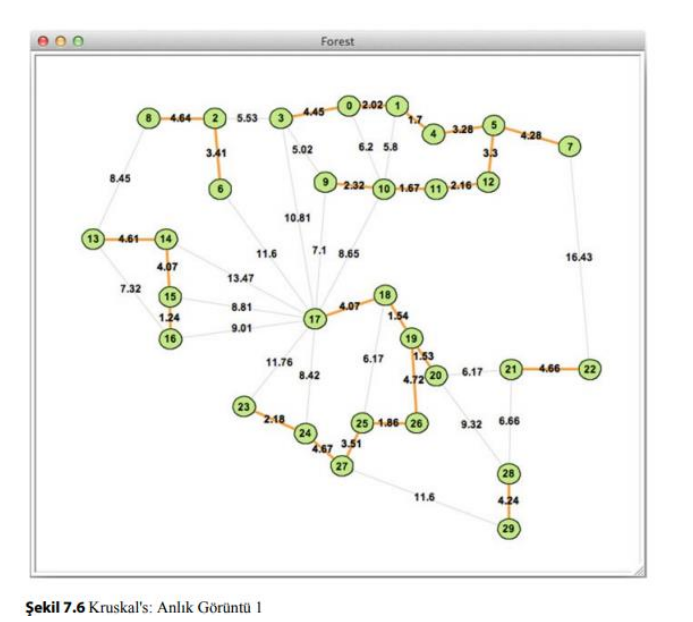
Algoritma, tüm kenarları ağırlıklarının artan sırasına göre sıralayarak başlar. Grafın tamamen bağlı olduğu varsayılırsa, yayılma tree |V|−1 kenar içerir. Algoritma, grafın tüm düğümleri için, başlangıçta yalnızca o düğümü içeren, her bir düğüm için bir küme oluşturarak başlar. Şekil **7.3**'teki örnekte, başlangıçta her biri bir düğüm içeren 30 küme bulunmaktadır.

Algoritma, spanning tree kenarları kümesine |V|−1 kenar eklenene kadar şu şekilde devam eder.

1. Sonraki en kısa kenar incelenir. Eğer kenarın iki uç noktasındaki iki vertex farklı kümelerde yer alıyorsa, o zaman kenar güvenle spanning tree kenarları kümesine eklenebilir. Yeni bir küme, iki vertex kümesinin birleşiminden oluşur ve önceki iki küme, vertex kümeleri listesinden çıkarılır.
2. Eğer kenarın iki uç noktasındaki iki vertex zaten aynı kümeye aitse, kenar göz ardı edilir.

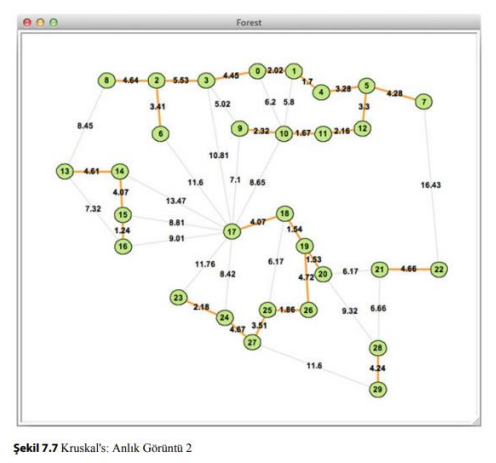
Bu, algoritmanın tamamıdır. Algoritma greedy (açgözlüdür) çünkü her zaman bir sonraki en küçük kenarı seçer, ta ki bunu yapmak bir döngü oluşturana kadar. Eğer bir kenar eklemek bir döngü oluşturacaksa, bu durum hemen fark edilir ve herhangi bir hatayı geri alıp geriye gitmeye gerek kalmaz.

Şekil **7.6**'yı göz önünde bulundurun. Bu anlık görüntüde algoritma zaten bir orman (forest) oluşturmuş, ancak henüz bir spanning tree (kapalı ağaç) oluşturmuş değil. Turuncu renkteki kenarlar, spanning tree'nin bir parçasıdır.

Bir sonraki en kısa kenar, vertex 3'ten vertex 9'a giden kenar şu anda inceleniyor. 3 ve 9'u içeren küme, {3, 0, 1, 4, 5, 12, 11, 10, 9} kümesidir. 3 ve 9 uç noktalarına sahip kenarın eklenmesi mümkün değildir çünkü vertex 3 ve 9 zaten aynı kümededir. Bu nedenle, bu kenar atlanır. Minimum ağırlıklı spanning tree' nin bir parçası olamaz.

Bir sonraki en kısa kenar, vertex 2 ile vertex 3 arasındaki kenardır. 2, {8, 2, 6} kümesinin bir elemanı olduğundan ve 3 ise önceki paragraftaki kümesinin bir elemanı olduğundan, {2, 3} kenarı minimum ağırlıklı spanning tree kenarlarına eklenir ve yeni küme {8, 2, 6, 3, 0, 1, 4, 5, 12, 11, 10, 9} oluşturulur, önceki iki alt kümesi yerine geçer. Bu durum Şekil **7.7**'de gösterildiği gibi olur.

Bir sonraki en kısa kenar, vertex 1 ile vertex 10 arasındaki kenardır. Bu kenar yine eklenemez çünkü 1 ve 10 aynı kümededir ve dolayısıyla kenarın eklenmesi bir döngüS oluşturur. Bir sonraki en kısa kenar, vertex 18 ile vertex 25 arasındaki kenardır, ancak yine eklenmesi bir döngü oluşturacağından bu kenar da atlanır. Algoritma, |V|−1 kenar ile sonuçlanan spanning tree oluşana kadar bu şekilde devam eder (grafın tam bağlantılı olduğu varsayılır).



**7.4.1 Doğruluk Kanıtı**

Kruskal algoritmasının doğru bir şekilde minimum ağırlıklı spanning tree bulduğunu kanıtlamak, çelişki yoluyla yapılabilir. Kanıt, spanning tree'de |V|−1 kenar olması gerektiğiyle başlar. Ardından, algoritma tarafından seçilen kenarlardan daha iyi olan başka bir kenarın spanning tree'ye eklenebileceğini varsayarız. Yeni kenar, bir ve yalnızca bir döngünün parçası olmalıdır. Eğer yeni kenarın eklenmesi iki veya daha fazla döngü oluşturuyorsa, bu durumda yeni kenar eklenmeden önce zaten tree'de bir döngü olmalıydı. Bu yeni oluşan döngüdeki kenarlardan birinin, minimum ağırlıklı spanning tree'den silinmesi gerekir, böylece tekrar bir tree elde edilir. Silinen kenarın, yeni eklenen kenardan daha büyük bir ağırlığa sahip olması gerekir. Bu yalnızca, yeni kenar ve silinen eski kenarın tam olarak aynı ağırlığa sahip olması durumunda mümkündür, çünkü döngüdeki tüm eski kenarlar, yeni kenar eklenmeden önce seçildi ve yeni kenar, onu seçmek bir döngü oluşturacağı için atlandı. Bu nedenle, eski kenarlardan aynı ağırlığa sahip olan kenarın düşürülmesi, aynı ağırlığa sahip bir minimum ağırlıklı spanning tree ile sonuçlanır. Bu durumda, yeni spanning tree, orijinal spanning tree ile aynı ağırlığa sahip olur ve bu, daha iyi bir kenarın bulunabileceği varsayımımızla çelişir.

**7.4.2 Kruskal'ın Karmaşıklık Analizi**

Kruskal algoritmasının karmaşıklığı, kenar listesinin sıralanmasına ve algoritma ilerledikçe kümelerin birleşiminin yapılmasına bağlıdır. Bir listeyi sıralamak, **4**. Bölümde quicksort'un karmaşıklığını incelediğimizde gösterildiği gibi, O(|E|log|E|) zaman karmaşıklığına sahiptir.

Listeyi sıralamak, Kruskal algoritmasının bir yarısıdır. Diğer yarısı ise doğru kenarları seçmektir. Hatırlayın ki, her kenar başta kendi başına bir küme içinde bulunur ve bir kenar, iki uç noktasındaki vertex'ler ayrı kümelerde ise minimum ağırlıklı spanning tree'ye ait olur. Eğer durum böyleyse, o zaman uç noktaları içeren iki küme birleşir ve bu iki kümenin birleşimi, ilerleyen adımlarda önceki iki kümenin yerini alır.

Bu algoritmanın bu kısmını uygulamak için üç işlem gereklidir:

1. İlk olarak, spanning tree'ye eklenmesi düşünülen kenarın her uç noktası için ait olduğu küme keşfedilmelidir.
2. Sonra, iki küme eşitlik açısından karşılaştırılmalıdır.
3. Son olarak, iki kümenin birleşimi oluşturulmalı ve gerekli güncellemeler yapılmalıdır, böylece iki uç nokta artık orijinal kümeleri yerine, bu iki kümenin birleşimine referans verir.

Bu işlemleri uygulamanın bir yolu, her pozisyonun grafikteki bir vertex'e karşılık geldiği bir küme listesi oluşturmaktır. Örneğin, metindeki örnekte vertex'ler 0-29 arasında numaralandırılmıştır, ancak vertex'ler, 0'dan başlayarak başka tamsayı kimlikleriyle yeniden atanabilir. Bir vertex'e karşılık gelen küme, O(1) zaman diliminde bulunabilir çünkü listede indeksleme, sabit zamanlı bir işlemdir.

Eğer her kümenin yalnızca bir kopyasının olduğundan emin olursak, iki kümenin aynı olup olmadığını O(1) zaman diliminde belirleyebiliriz. Bunun için, küme referanslarını karşılaştırarak aynı küme olup olmadıklarını kontrol edebiliriz. Python'da is anahtar kelimesi bu işlemi gerçekleştirecektir. Yani, eğer x ve y'nin aynı kümeye referans verip vermediğini öğrenmek istiyorsak, x is y şeklinde yazabiliriz ve bu işlem O(1) zaman karmaşıklığına sahiptir.

Üçüncü işlem, önceki iki kümeden yeni bir küme oluşturmayı gerektirir. Bu işlem, |V|−1 kez gerçekleştirilecektir. En kötü durumda, bu işlem ilk kez yapıldığında 1 vertex mevcut bir kümeye eklenir. İkinci kez yapıldığında, mevcut bir kümeye 2 vertex eklenir ve bu şekilde devam eder. Sonuç olarak, bu işlemin genel en kötü durum karmaşıklığı O(|V|²) olacaktır, yine grafın bağlantılı olduğu varsayılırsa. Açıkça, bu algoritmanın pahalı işlemi budur. Bir sonraki bölüm, bu işlemi önemli ölçüde iyileştiren bir veri yapısını sunmaktadır.

**7.4.3 Bölme Veri Yapısı (Partition Data Structure)**

Üçüncü gerekli işlemi, yani iki kümenin birleştirilerek tek bir küme oluşturulmasını iyileştirmek için Bölme (Partition) adı verilen özel bir veri yapısı kullanılabilir.

Bölme veri yapısı, her düğüm (vertex) için bir giriş içeren bir tamsayı listesi barındırır. Başlangıçta, bu liste düğüm indisleriyle eşleşen tamsayıları içerir:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29]

Bu listeyi, şu ana kadar oluşturulmuş genişleyen ormanda (spanning forest) bulunan bağlantılı kenar kümelerini temsil eden bir ağaçlar listesi olarak düşünebiliriz.

* Bir ağacın kökü (root), listedeki belirli bir konumdaki değerin, o konumun indisiyle eşleştiği durumunda gösterilir.
* Başlangıçta her düğüm kendi ağacının köküdür, yani her düğüm kendi kümesinde bulunur.

**Bir Düğümün Kümeye Aitliğini Keşfetme**

Bir düğümün hangi kümeye ait olduğunu bulmak, o düğümün ağacının köküne kadar takip edilmesi anlamına gelir.

Örneğin, 3. Düğümden (vertex’ten) 9. Düğüme(vertex’e) olan kenarın minimum ağırlıklı genişleyen ağaca eklenme ihtimali değerlendirilirken, o andaki bölme yapısı şu şekildedir:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

[ 4, 4, 2, 7, 5,11, 2, 7, 2,11,117,11,16,16,16,16,19,19,19,19,22,22,24,26,26,19,26,29,29]

* Düğüm(vertex) 3, şu an kendi ağacının kökü değildir çünkü indeks 3’te 7 değeri bulunur.
* Bu yüzden indeks 7'ye bakılır, burada da 7 bulunduğu için 3. Düğümün(vertex’in) ağacının kökü 7'dir.
* 9. Düğümün(vertex) kökünü bulmak için, indeks 9’da 11 değeri bulunur.
* İndeks 11’de 7 değeri yer alır, yani 9. Düğümün(vertex’in) kökü de 7'dir.

Sonuç olarak, 3. ve 9. Düğümler(vertex) zaten aynı kümede olduğu için 3 ile 9 arasındaki kenar eklenemez, çünkü bir döngü (cycle) oluşacaktır.

**Yeni Kenar Eklenmesi ve Küme Birleştirme**

Bir sonraki ele alınan kenar, 2. ve 3. düğümler arasındaki kenardır.

* 2. Düğümün(vertex’in) kökü, indeks 2'deki değerdir, yani 2'dir.
* 3. Düğümün(vertex’in) kökü ise daha önce bulunduğu gibi 7'dir.

Bu iki düğüm farklı kümelerde olduğu için bu kenar minimum ağırlıklı genişleyen ağaca eklenebilir.  
 Şimdi, bu iki kümeyi birleştirme işlemi gerçekleştirilir:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

[4, 4, 2, 2, 5,11, 2, 2, 2,11,11,7,11,16,16,16,16,19,19,19,19,22,22,24,26,26,19,26,29,29]

Bu noktada 7 köklü ağaç, artık 2 köklü ağaç olarak değiştirildi.  
 İki kümeyi birleştirmek için bir ağacın kökünü diğer ağacın köküne işaret ettirmek yeterlidir.

**Bölme Veri Yapısında İşlem ve Karmaşıklık Analizi**

sameSetAndUnion adlı yöntem, birleştirme ve küme denetleme işlemlerini bir arada yapar.  
 Bu yöntem iki düğüm numarası alır:

* Eğer aynı kümeye aitlerse, true döndürülür.
* Aynı kümeye ait değillerse, bir ağacın kökü diğerine bağlanır ve false döndürülür.

Bu yöntemde:

* Kök bulma işlemi en kötü durumda O(|V|) zaman alabilir.
* Ancak uygulamada bu ağaçlar oldukça düz yapıda olur.
  + Örneğin, bu bölümdeki örnekte ortalama küme derinliği 1.7428 olarak hesaplanmıştır.
  + yani minimum ağırlıklı yayılan ağaca eklemeyi düşünmek için 30 köşeli ve 45 kenarlı bir grafikte bir küme ağacının kökünü bulmak ortalama 1.7428 karşılaştırma gerektirir.
  + 133 düğümlü ve 8778 kenarlı başka bir grafikte ortalama derinlik 7.5656'dır. Bu aynı SetAndUnion yönteminin ortalama karmaşıklığı, Bölüm 7.4.2'de ele alınan çözümden çok daha iyidir.
* SameSetAndUnion’ ın ortalama durum karmaşıklığı O(log|V|) seviyesine yaklaşır.

Bu nedenle, Kruskal Algoritmasının ikinci kısmı, Bölme Veri Yapısı kullanıldığında O(|E|log|V|) karmaşıklığına sahiptir.

**Kruskal Algoritmasının Genel Karmaşıklık Analizi**

Bağlı bir grafikte:

* Kenar sayısı düğüm sayısından en az bir eksik olmalıdır.
* Kenarları sıralamak O(|E|log|E|) zaman alır.
* Kruskal ’ın ikinci adımı O(|E|log|V|) zaman alır.

Sonuç olarak:

* Bağlı bir grafikte O(|E|log|V|) ≤ O(|E|log|E|) olduğu için, Kruskal Algoritmasının ortalama zaman karmaşıklığı O(|E|log|E|) olarak belirlenebilir.
* Kruskal Algoritması oldukça verimlidir ve büyük grafikleri bile hızlı bir şekilde işler.

Sonuç

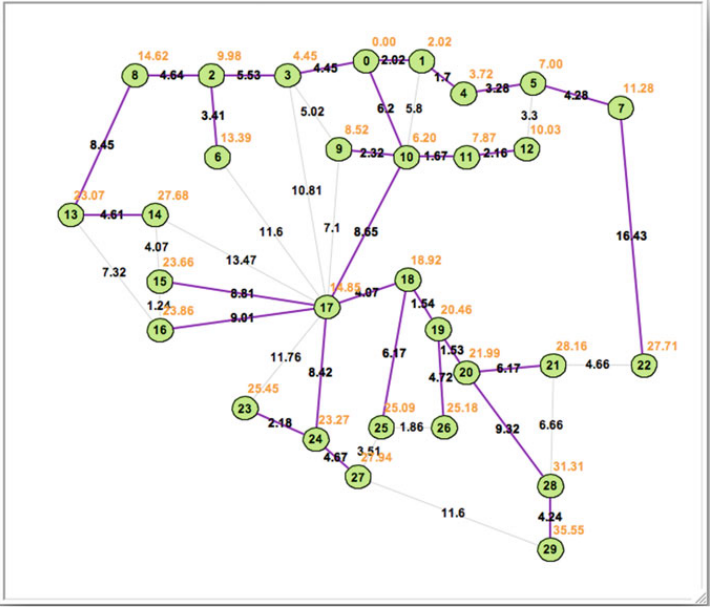
* Bölme veri yapısı, Kruskal Algoritmasındaki küme birleştirme işlemini verimli hale getirir.
* sameSetAndUnion yöntemi sayesinde, Kruskal ’ın ikinci aşaması O(|E|log|V|) karmaşıklığında çalışır.
* Bu sayede algoritma büyük ölçekli grafikleri hızlıca işleyebilir.

Böylece, Kruskal Algoritması en küçük ağırlıklı genişleyen ağacı (MST) etkin şekilde bulur.

───────────────────────────────────────────────────────────────────────────

**7.5 Dijkstra Algoritması**

Edsger Dijkstra, 1930-2002 yılları arasında yaşamış Hollandalı bir bilgisayar bilimcisiydi.  
 1959 yılında yayımladığı kısa bir makalede [4], Kruskal ’ın minimum kapsayan ağaç problemine getirdiği çözüme değinmiş ve bazı durumlarda daha verimli olabilecek alternatif bir yöntem önermiştir. Daha da önemlisi, belirli bir noktadan diğer tüm noktalara en kısa yolları bulan bir algoritma geliştirmiştir..



**Şekil 7.8** Kaynak Vertex 0'dan Minimum Maliyetli Yollar ve Toplam Maliyet

Herhangi iki düğüm arasındaki minimum maliyet yolunu bulmak için kullanılan bu algoritma, bazen bir kaynak düğümünden tüm diğer düğümlere kadar olan minimum maliyet yolunu bulacak şekilde genellenebilir. Bu algoritma, Dijkstra algoritması olarak bilinir. Şekil 7.8, Dijkstra algoritmasının, Şekil 7.3'te ilk olarak sunulan grafikte çalıştırılmasının sonucunu göstermektedir. Mor kenarlar, kaynak düğüm 0'dan diğer tüm düğümlere kadar olan minimum maliyet yollarını göstermektedir. Turuncu değerler ise kaynak düğüm 0'dan her bir düğüme ulaşmanın minimum maliyetini göstermektedir.

Bir düğümden diğerine minimum maliyet yolunu verimli bir şekilde bulmak, ağ yönlendirme, gezi planlaması ve düğümlerin ara hedefleri temsil ettiği, kenarların ise bu ara hedefler arasında geçiş maliyetlerini temsil ettiği diğer planlama problemleri de dahil olmak üzere pek çok farklı problemde kullanılmaktadır. Bu tür planlama problemleri oldukça yaygındır.

Dijkstra algoritması, tek bir kaynak düğümünden açgözlü bir şekilde ilerler. Grafikteki her bir düğüm, kaynak düğümünden o düğüme kadar olan yolun ağırlıklı kenarlarının toplamı olan bir maliyete atanır. Başlangıçta kaynak düğümüne 0 maliyet atanır. Diğer tüm düğümlere başlangıçta sonsuz maliyet atanır. Grafikteki tüm kenarların ağırlıklarının toplamından büyük herhangi bir değer sonsuz olarak kabul edilebilir.

Dijkstra algoritması, derinlik öncelikli arama (DFS) ile bazı benzerlikler gösterir. Algoritma, derinlik öncelikli arama gibi, bir kaynakla başlayarak sonunda grafikteki her düğümü ziyaret eder. Dijkstra algoritması iki küme kullanır. Birincisi, henüz minimum maliyet yolları aranması gereken düğümlerin bulunduğu ziyaret edilmemiş kümedir. Bu küme, derinlik öncelikli arama yaparken kullanılan yığın gibi işlev görür. Ziyaret edilmiş küme ise algoritmanın kullandığı diğer kümedir. Bu küme, minimum maliyet ve yolu hesaplanmış olan tüm düğümleri içerir ve derinlik öncelikli aramada kullanılan ziyaret edilmiş küme ile aynı işlevi görür.

Kaynaktan bir vertex'e (verteks) (v) giden minimum maliyetli yolu takip etmek için, sadece v'ye giden yoldaki bir önceki vertex'i takip etmek gerekir. Her bir vertex (v) için, kaynaktan gelen yoldaki bir önceki vertex'i takip ederiz.

Başlangıçta, 0 maliyeti olan kaynak düğüm, ziyaret edilmemiş kümeye eklenir. Daha sonra algoritma şu şekilde ilerler, ziyaret edilmemiş kümede en az bir düğüm kaldığı sürece:

1. Ziyaret edilmemiş kümeden en düşük maliyeti olan "current" adlı düğüm çıkarılır. Bu düğüme giden tüm diğer yollar daha yüksek maliyete sahip olmalıdır, çünkü aksi takdirde bu yollar daha düşük maliyetle ziyaret edilmemiş kümede olurdu.
2. "Current" düğümü, ziyaret edilmiş kümeye eklenir.
3. Herhangi bir "adjacent" (komşu) düğümü, "current" ile komşu olan her düğüm için, bu düğümün ziyaret edilmiş kümede olup olmadığı kontrol edilir. Eğer "adjacent" ziyaret edilmiş kümede ise, o düğüme kaynaktan ulaşmanın minimum maliyetini zaten biliyoruz ve hiçbir şey yapılmaz.
4. Eğer "adjacent" ziyaret edilmiş kümede değilse, "current" ile "adjacent" arasındaki kenar olan e'yi geçerek "adjacent"e ulaşmanın yeni bir maliyeti hesaplanır. Bu yeni maliyet, "current"e ulaşmanın maliyeti ile e'nin ağırlığının toplamı olarak bulunabilir. Eğer bu yeni maliyet, "adjacent"e ulaşmanın mevcut maliyetinden daha iyi bir maliyetse, "adjacent"in maliyeti güncellenir ve "current", "adjacent"in önceki düğümü olarak kaydedilir. Ayrıca "adjacent" ziyaret edilmemiş kümeye eklenir.

Bu algoritma tamamlandığında, tüm düğümlere ulaşma maliyeti hesaplanmış olur, ancak bu, tüm düğümlerin kaynak düğümünden erişilebilir olduğu varsayımıyla yapılır. Ayrıca, her düğüme olan minimum maliyet yolunu belirlemek için algoritma çalıştırılırken tutulan önceki düğüm bilgisi kullanılabilir.

**7.5.1 Dijkstra'nın Zaman Karmaşıklığı Analizi**

Dijkstra'nın algoritmasının ilk adımında, bir sonraki geçerli tepe noktası her zaman en küçük maliyete sahip ziyaret edilmemiş tepe noktasıdır. Her zaman şimdiye kadarki en düşük maliyetli tepe noktasını seçerek, bu tepe noktasına giden daha ucuz başka bir yol olmadığından emin olabilir, her zaman grafikteki en ucuz yolları bulmak için aramamızda bir sonraki en ucuz tepe noktasını dikkate alarak ilerleyin.

Basit, yönlendirilmemiş bir grafikteki herhangi bir köşenin kenar sayısı her zaman grafikteki toplam köşe sayısından az olacaktır. Her köşe, 1. adımdaki algoritmada tam olarak bir kez geçerli köşe olur. Bir sonraki akımı bulmanın O(|V|) zaman aldığını varsayalım. Bu işlem |V| kez gerçekleştiğinden, ilk adımın karmaşıklığı algoritmayı çalıştırma süresince O(|V|²) 'dir. Adımların geri kalanı akıma komşu olan kenarları dikkate alır. Basit, yönlendirilmemiş bir grafikteki herhangi bir tepe noktasının kenar sayısı her zaman |V| 'den az olacağından, algoritmanın geri kalanı O(|V|²) 'den daha kısa sürede çalışır zaman. Dolayısıyla, Dijkstra Algoritmasının karmaşıklığı, ilk adımın bir sonraki geçerli vertex'i bulmak için O(|V|²) sürdüğünü varsayarsak O(|V|)’ dir. Ziyaret edilmemiş kümemiz için bir öncelik kuyruğu kullanırsak, bir sonraki akımın seçilmesinin O(log|V|) zamanda yapılabileceği ortaya çıkmaktadır. Bir öncelik kuyruğu kullanıldığında, Dijkstra Algoritması O(|V|log|V|) zamanda çalışacaktır.

**7.6 - Grafik Gösterimleri**

Bir grafiğin G = (V,E) bir programda nasıl temsil edileceği, programın ne yapması gerektiğine bağlıdır. Şekil 7.2'deki yönlendirilmiş grafiği göz önünde bulundurun. Grafik, Şekil 7.6.1'de gösterildiği gibi, düğümler ve kenarları içeren bir XML dosyasında saklanabilir. Ağırlıklı bir grafik, grafikteki her kenar için bir ağırlık özelliği içerir. Bu XML dosyası formatında, kenarlar hangi düğümlere bağlı olduklarını göstermek için vertexId'yi kullanır. Şekil 7.2'de görünen etiketler yalnızca etiketlerdir ve kenarları düğümlerle ilişkilendirmek için XML dosyasında kullanılmaz

**7.6.1 Bir XML Dosyası**

<?xml version="1.0" encoding="UTF-8"?>

<Graph width="595.80" height="229.20" directed="True" weighted="False">

<Vertices>

<Vertex vertexId="12" x="343.15" y="156.10" label="10"/>

<Vertex vertexId="11" x="246.15" y="161.10" label="9"/>

<Vertex vertexId="10" x="288.15" y="58.10" label="0"/>

<Vertex vertexId="9" x="374.15" y="58.10" label="1"/>

<Vertex vertexId="8" x="135.15" y="156.10" label="6"/>

<Vertex vertexId="7" x="49.65" y="83.10" label="2"/>

<Vertex vertexId="6" x="167.15" y="83.05" label="3"/>

<Vertex vertexId="5" x="121.15" y="19.10" label="8"/>

<Vertex vertexId="4" x="419.15" y="204.10" label="11"/>

<Vertex vertexId="3" x="426.15" y="87.10" label="4"/>

<Vertex vertexId="2" x="546.15" y="96.10" label="5"/>

<Vertex vertexId="1" x="546.15" y="210.10" label="7"/>

<Vertex vertexId="0" x="485.15" y="161.10" label="12"/>

</Vertices>

<Edges>

<Edge tail="12" head="10"/>

<Edge tail="10" head="6"/>

<Edge tail="6" head="11"/>

<Edge tail="7" head="6"/>

<Edge tail="7" head="8"/>

<Edge tail="7" head="5"/>

<Edge tail="11" head="12"/>

<Edge tail="12" head="4"/>

<Edge tail="4" head="0"/>

<Edge tail="0" head="2"/>

<Edge tail="2" head="1"/>

<Edge tail="2" head="3"/>

<Edge tail="3" head="9"/>

<Edge tail="9" head="10"/>

<Edge tail="9" head="12"/>

</Edges>

</Graph>

Düğümün x ve y koordinatları, herhangi bir grafik temsili için zorunlu değildir, ancak bir grafiği çizmek için düğümlerin konum bilgisine sahip olmak faydalıdır. Tüm bu bilgiler XML dosyasında saklanır, ancak bu bölümde sunulan üç algoritma için ne tür bilgiler gereklidir?

Bir grafikte derinlik öncelikli arama (DFS) yaparken, arama ilerledikçe düğümler bir yığına (stack) yerleştirilir. Bu durumda, arama için düğüm bilgileri saklanmalıdır. Bu durumda, kenarların uç noktalarındaki vertexId'leri bulunduğundan, grafikte düğümleri hızlıca arayabilmek için bir yöntem olması faydalı olacaktır. Düğüm 'den düğümlere bir harita ya da sözlük kullanmak pratik olur. Bu yüzden, düğüm bilgilerini tutacak bir sınıf oluşturmak mantıklı olacaktır, tıpkı 7.6.2 Bölümündeki sınıf tanımı gibi.

**7.6.2 Düğüm Sınıfı (Vertex Class)**

1. class Vertex:

2. def **init**(self,vertexId,x,y,label):

3. self.vertexId = vertexId

4. self.x = x

5. self.y = y

6 . self.label = label

7. self.adjacent = []

8 . self.previous = None

Bu Düğüm sınıfı tanımında, yönlendirilmiş grafikler için kenarları düğümle birlikte saklamak mantıklıdır, çünkü kenarlar düğümleri birbirine bağlar. Komşu liste, komşu düğümlerin listesini tutabilir. Derinlik öncelikli arama (DFS) çalıştırırken, grafikteki her bir düğüm için vertexId ile Vertex arasında bir harita, algoritma için gerekli olan bilgiyi sağlar.

Kruskal algoritmasını uygularken, grafikteki önemli özellik kenarların bir listesidir. 7.6.3 Bölümü'ndeki sınıf tanımı, kenar nesnelerinin sıralanmasını sağlayan bir "küçükten büyüğe" (less-than) metoduna sahiptir, bu da Kruskal algoritması için çok önemlidir. Algoritma için düğümler gereksizdir. Kenarların bir listesi ve bölme veri yapısı, Kruskal algoritmasını çalıştırmak için yeterlidir.

**7.6.3 Bir Kenar sınıfı**

class Edge:

def init(self,v1,v2,weight=0):

self.v1 = v1

self.v2 = v2

self.weight = weight

def **lt**(self,other):

return self.weight < other.weight

Dijkstra algoritmasının çalıştırılması, hem kenar (edge) hem de düğüm (vertex) nesnelerine sahip olmaktan fayda sağlar. Algoritma, her kenarın ağırlığını kullanır, bu nedenle ağırlıkları kenarlarda saklamak ve düğümler ile kenarları ilişkilendirmek yararlıdır.

Grafikleri temsil etmek için başka yöntemler de vardır. Örneğin, iki boyutlu bir matris kullanılarak düğümler arasındaki kenarlar temsil edilebilir. Bu matrisin satır ve sütunları düğümleri temsil eder. vi düğümünden vj düğümüne olan bir kenarın ağırlığı, matrix[i][j] konumunda saklanır. Bu tür bir gösterim bitişik matris (adjacency matrix) olarak adlandırılır. Ancak, bitişik matrisler genellikle seyrek (sparse) olur ve fazla yer kapladıkları için pratikte pek fazla kullanılmazlar.

Seçilen grafik gösterimi, yapılan işe bağlıdır.

* Düğümlerin sadece komşuluk bilgisi içermesi yeterli olabilir.
* Kruskal algoritması için kenar listesi (edge list) yeterlidir.
* Dijkstra algoritması, hem düğüm hem de kenar bilgisine ihtiyaç duyar.
* Bazı durumlarda bitişik matrisler kullanılabilir.

Bir programcı olarak, boşa harcanan alanı, algoritmaların ihtiyaçlarını ve verimliliğini dikkate almalıyız. Seçtiğimiz veri gösteriminin, programlarımız üzerindeki etkilerini iyi değerlendirmemiz gerekir.

**7.7 Bölüm Özeti**

Bu bölümde grafikler ve bunlara ilişkin çeşitli algoritmalar ele alındı. Aşağıdaki temel konular üzerinde duruldu:

* **Grafik Tanımları ve Türleri:**
  + Grafikler düğümler (vertices) ve kenarlardan (edges) oluşur.
  + Grafikler yönlendirilmiş (directed) veya yönlendirilmemiş (undirected) olabilir.
  + Ağaçlar (trees): Her iki düğüm arasında yalnızca bir yol bulunan grafiklerdir.
  + Örtücü Ağaç (Spanning Tree): Bağlantılı bir grafikte tüm düğümleri içeren alt grafiktir.
  + Minimum Ağırlıklı Örtücü Ağaç (Minimum Spanning Tree - MST): Kruskal algoritması kullanılarak bulunur.
* Grafik Algoritmaları:
  + Derinlik Öncelikli Arama (DFS): Yığıt (stack) veri yapısını kullanarak grafiği keşfeder.
  + Kruskal Algoritması: En düşük ağırlıklı kenarları seçerek minimum ağırlıklı örtücü ağacı oluşturur.
  + Dijkstra Algoritması: Belirtilen bir başlangıç düğümünden tüm düğümlere en düşük maliyetli yolları bulur.
* Grafik Temsilleri:
  + Komşuluk Listesi (Adjacency List): Her düğümün komşularını liste olarak saklar.
  + Kenar Listesi (Edge List): Grafiğin tüm kenarlarını liste olarak saklar. Kruskal algoritması için idealdir.
  + Komşuluk Matrisi (Adjacency Matrix): Satır ve sütunları düğümleri temsil eden, kenar ağırlıklarını içeren matristir. Ancak gereksiz bellek kullanımı nedeniyle pratikte daha az tercih edilir.
* Grafik Temsili Seçimi:
  + Kruskal algoritması için kenar listesi yeterlidir.
  + Dijkstra algoritması için düğümler ve kenar bilgileri gereklidir.
  + Derinlik öncelikli arama (DFS) için düğüm listesi ve komşuluk bilgileri gereklidir.
  + Bellek ve algoritmanın gereksinimlerine bağlı olarak uygun grafik temsili seçilmelidir.

Bu bölümde grafiklerin ve algoritmaların temelleri ele alınarak, verimli veri yapıları seçiminin algoritma performansı üzerindeki etkileri vurgulanmıştır

**7.8 İnceleme Soruları**

Aşağıdaki kısa cevap, çoktan seçmeli ve doğru/yanlış sorularını yanıtlayarak bu bölümü ne kadar anladığınızı test edin.

1. Bir grafiğin tanımında G = (V, E) ifadesinde V ve E neyi temsil eder?
2. Yönlendirilmiş (directed) ve yönlendirilmemiş (undirected) grafiklerde E kümesinin tanımı arasındaki fark nedir?
3. Derinlik öncelikli aramada (Depth First Search - DFS) ziyaret edilenler (visited) kümesinin amacı nedir?
4. Derinlik öncelikli aramada geri izleme (backtracking) nasıl gerçekleştirilir? Geri izleme süreci nasıl işler?
5. Bir grafikte bir yol (path) nedir ve bir döngüden (cycle) nasıl farklıdır?
6. Bir ağaç (tree) nedir? Şekil 7.2’de verilen grafikte, 0, 1 ve 10 düğümlerini içeren üç farklı ağaç gösterin.
7. Kruskal algoritması, minimum ağırlıklı kapsam ağacını (minimum weighted spanning tree) oluştururken neden hiçbir zaman hata yapmaz?
8. Dijkstra algoritması, düğümlere olan yolların maliyetini hesaplarken neden hiçbir zaman hata yapmaz?
9. Kruskal algoritması için en uygun grafik gösterimi hangisidir? Neden?
10. Dijkstra algoritması neden önceki düğümü saklar? Önceki düğümün saklanmasının amacı nedir?

**7.9 Programlama Sorunları**

1. Bir program yazın: Grafikte Şekil 7.9'da gösterilen 9 ve 29 numaralı düğümler (vertex) arasında bir yol bulsun. Yolun (yani, geçilmesi gereken düğüm dizisinin) yazdırıldığından emin olun. Bu grafiği tanımlayan bir XML dosyası, kitabın web sitesinde bulunabilir.
2. İlk problemi değiştirerek: 9 ve 29 numaralı düğümler arasındaki kenar (edge) sayısı bakımından en kısa yolu bulun. Başka bir deyişle, kenar ağırlıklarını yok sayarak çözüm üretin. Bu çözüm için Genişlik Öncelikli Arama (Breadth-First Search - BFS) algoritmasını kullanın.
3. Şekil 7.9’daki grafikte Dijkstra algoritmasını uygulayarak, 9 numaralı düğümden diğer tüm düğümlere gitmenin en düşük maliyetini bulan bir program yazın.
4. Şekil 7.9’daki yönlendirilmiş (directed) grafikte veya bölümdeki yönlendirilmemiş (undirected) grafikte Kruskal algoritmasını uygulayan bir program yazın. Her iki grafiğin XML dosyaları, kitabın web sitesinde bulunabilir.
5. Her grafiğin açıkça temsil edilmesi gerekmez. Bazen, bir düğüm verildiğinde bu düğüme bitişik (komşu) düğümleri hesaplayan bir fonksiyon yazmak yeterlidir. Örneğin, su kovası problemini ele alalım:

> Bu problemde biri 3 galonluk, diğeri 5 galonluk olmak üzere iki kova vardır.

> Göreviniz, 5 galonluk kovaya tam olarak 4 galon su koymaktır.

> Oyun kuralları şunlardır:

Bir kovayı tamamen doldurabilirsiniz.

Bir kovadan diğerine su dökebilirsiniz.

Bir kovayı tamamen boşaltabilirsiniz.

Ancak, kovayı kısmen dolduramazsınız.

Görev:

İki boş kova ile başlayıp 5 galonluk kovada 4 galon su elde etmek için bir program yazın.

Derinlik Öncelikli Arama (Depth-First Search - DFS) algoritmasını kullanarak problemi bir grafik olarak temsil edin.

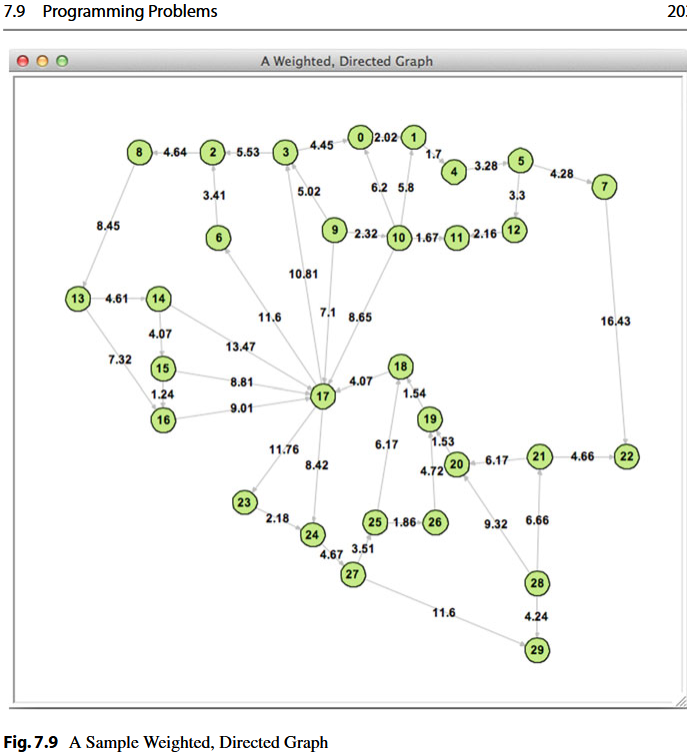
Düğümler, her iki kovadaki su miktarını içeren durumları temsil eder.

Komşu (adjacent) fonksiyonu, bir durumdan ulaşılabilecek geçerli durumların listesini döndürmelidir.

Gereksiz durumları önlemek için, örneğin, 5 galonluk kovada 6 galonluk su gibi geçersiz durumları filtreleyin.

Program, kovaların başlangıçta boş olduğu durumdan 5 galonluk kovada 4 galon su elde edene kadar yapılması gereken adımları yazdırmalıdır.

mutlaka en kısa çözüm olmayabilir, ancak geçerli bir çözüm olmalıdır.



1. Bir iki parçalı (bipartite) grafiği test eden bir program yazın.

Bir grafik, düğümlerinin iki kümeye ayrılabileceği ve aynı kümedeki iki düğüm arasında kenar bulunmadığı durumlarda iki parçalıdır.

Grafikteki tüm kenarlar, farklı kümelerdeki düğümler arasındadır.

Grafiğin iki parçalı olup olmadığını anlamak için DFS gibi bir gezinti algoritması kullanarak tekil (odd) döngüleri arayın.

Bir grafik, yalnızca tekil döngü içermiyorsa iki parçalıdır.

Program, "Evet, iki parçalıdır." veya "Hayır, iki parçalı değildir." şeklinde çıktı

vermelidir.

1. Önceki problemi geliştirerek, grafik iki parçalıysa her iki kümenin düğümlerini yazdıran bir program yazın.