**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, 2,11,11, 7,11,16,16,16,16,19,19,19,19,22,22,24,26,26,19,26,29,29]

* 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’ler) 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’nın Algoritması**

metin, ekran görüntüsü, diyagram, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.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.

7.5 Dijkstra’nınAlgoritması Şekil 7.8: Kaynak düğüm 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**

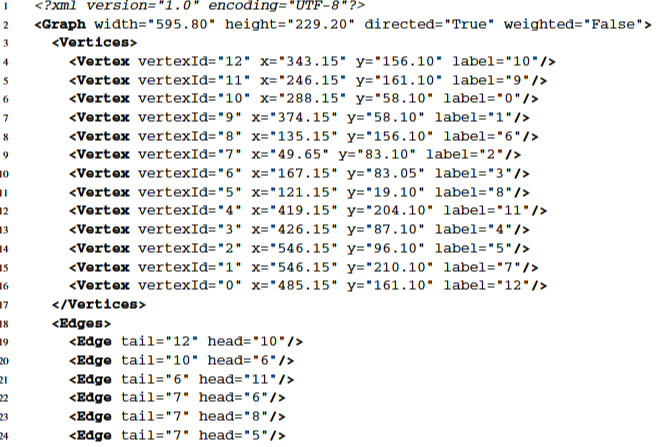
Dijkstra algoritmasının ilk adımında, bir sonraki mevcut düğüm her zaman en küçük maliyete sahip ziyaret edilmemiş düğümdür. Şu ana kadar en küçük maliyete sahip düğümü her zaman seçerek, bu düğüme başka hiçbir ucuz yolun olmadığından emin olabiliriz, çünkü her zaman grafikteki en ucuz yolları bulmak için aramamızda bir sonraki en ucuz düğümü dikkate alıyoruz.

Basit, yönsüz bir grafikte herhangi bir düğümün kenar sayısı, grafikteki toplam düğüm sayısından her zaman daha az olacaktır. Algoritmada, her düğüm tam olarak bir kez mevcut düğüm olur. Bir sonraki mevcut düğümün bulunmasının O(|V|) zaman aldığı varsayalım. Bu işlem, |V| kez gerçekleştiği için, ilk adımın karmaşıklığı O(|V|²) olur. Geriye kalan adımlar, mevcut düğüme komşu olan kenarları dikkate alır. Basit, yönsüz bir grafikte herhangi bir düğümün kenar sayısı her zaman |V| ’den daha az olacağı için, algoritmanın geri kalanı O(|V|²)'den daha az bir zaman alır. Yani, Dijkstra algoritmasının karmaşıklığı, ilk adımın bir sonraki mevcut düğümü bulmanın O(|V|) zaman aldığı varsayımıyla O(|V|²)’dir.

Gerçekten de, bir sonraki mevcut düğümü seçmek, ziyaret edilmemiş küme için bir öncelik sırası kuyruğu kullanarak O(log|V|) zamanında yapılabilir. Öncelik sırası kuyrukları ve bunların implementasyonları 9. Bölümde tartışılmaktadır. Bir öncelik sırası kuyruğu kullanarak, Dijkstra algoritması O(|V|log|V|) zamanında çalışacaktır.

**7.6 Grafik Temsilleri**  
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 Grafiğin XML Dosyası**

metin, yazı tipi, ekran görüntüsü, beyaz içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

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 Bir Düğüm Sınıfı (Vertex Class)**

metin, ekran görüntüsü, yazı tipi, cebir içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

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**

metin, yazı tipi, ekran görüntüsü, makbuz içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

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: Grafikler ve Algoritmalar**

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 Gözden Geçirme 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 Problemleri**

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.

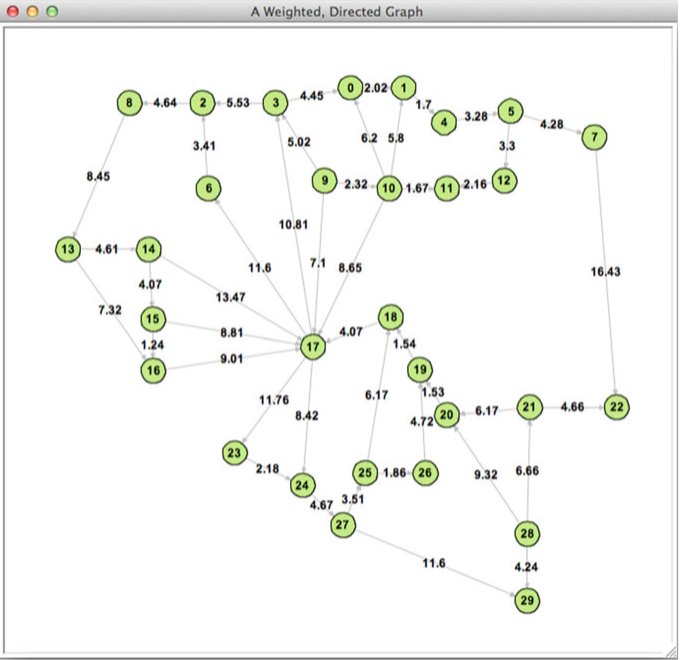
> Göreviniz, 5 galonluk kovaya tam olarak 4 galon su koymaktır.

> 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.
* Çözüm mutlaka en kısa çözüm olmayabilir, ancak geçerli bir çözüm olmalıdır.



**6.** 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.

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