7.4 Kruskal Algoritması

7.4.1 Doğruluk Kanıtı

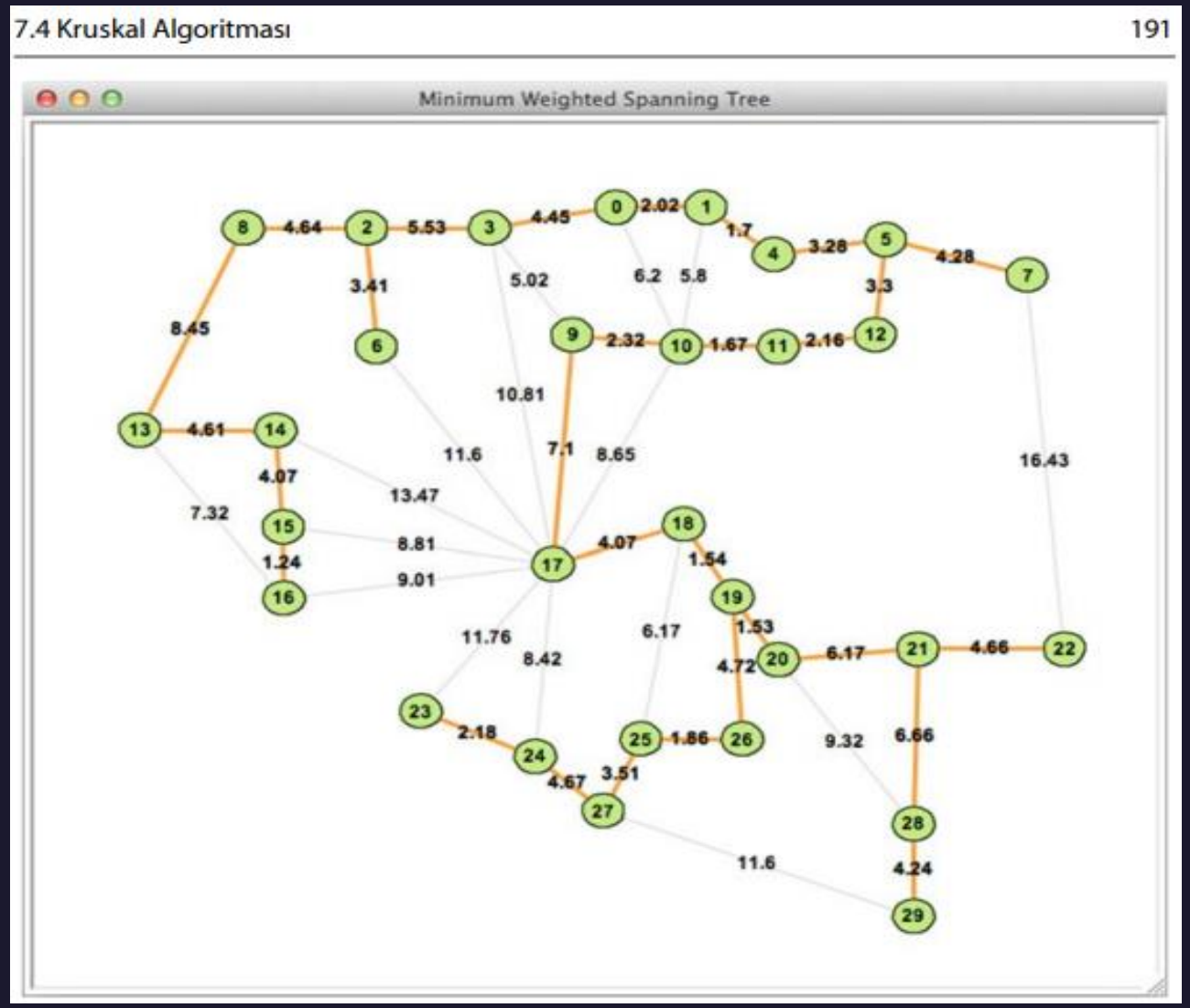
7.4.2 Kruskal'ın Karmaşıklık Analizi

7.4.3 Bölme Veri Yapısı

7.4 Kruskal Algoritması

- Bir an için kışın yolların kardan temizlenmesinden sorumlu olan ancak beklenmedik miktarda kar yağması nedeniyle parası tükenen bir ilçeyi düşünün. İlçe yöneticisine kışın geri kalanında sadece gerekli yolları küreyerek maliyetleri düşürmesi söylenmiştir. Amir, herhangi bir kişinin ilçedeki bir noktadan başka bir noktaya seyahat edebilmesi için sürülmesi gereken toplam milin en kısa sayısını bulmak istiyor, ancak bunun en kısa yoldan olması gerekmiyor. İlçe yöneticisi, ilçede ihtiyaç duyduğunuz her yere ulaşabileceğinizi garanti ederken, sürülen yolların kilometresini en aza indirmek ister. Joseph Kruskal 1928-2010 yılları arasında yaşamış Amerikalı bir bilgisayar bilimcisi ve matematikçiydi. Bu problemi hayal etmiş, ağırlıklı bir çizge cinsinden resmileştirmiş ve bu problemi çözmek için bir algoritma geliştirmiştir. Algoritması ilk olarak Proceedings of the American Mathematical Society'de yayımlanmıştır [5] ve genellikle Kruskal Algoritması olarak adlandırılır.

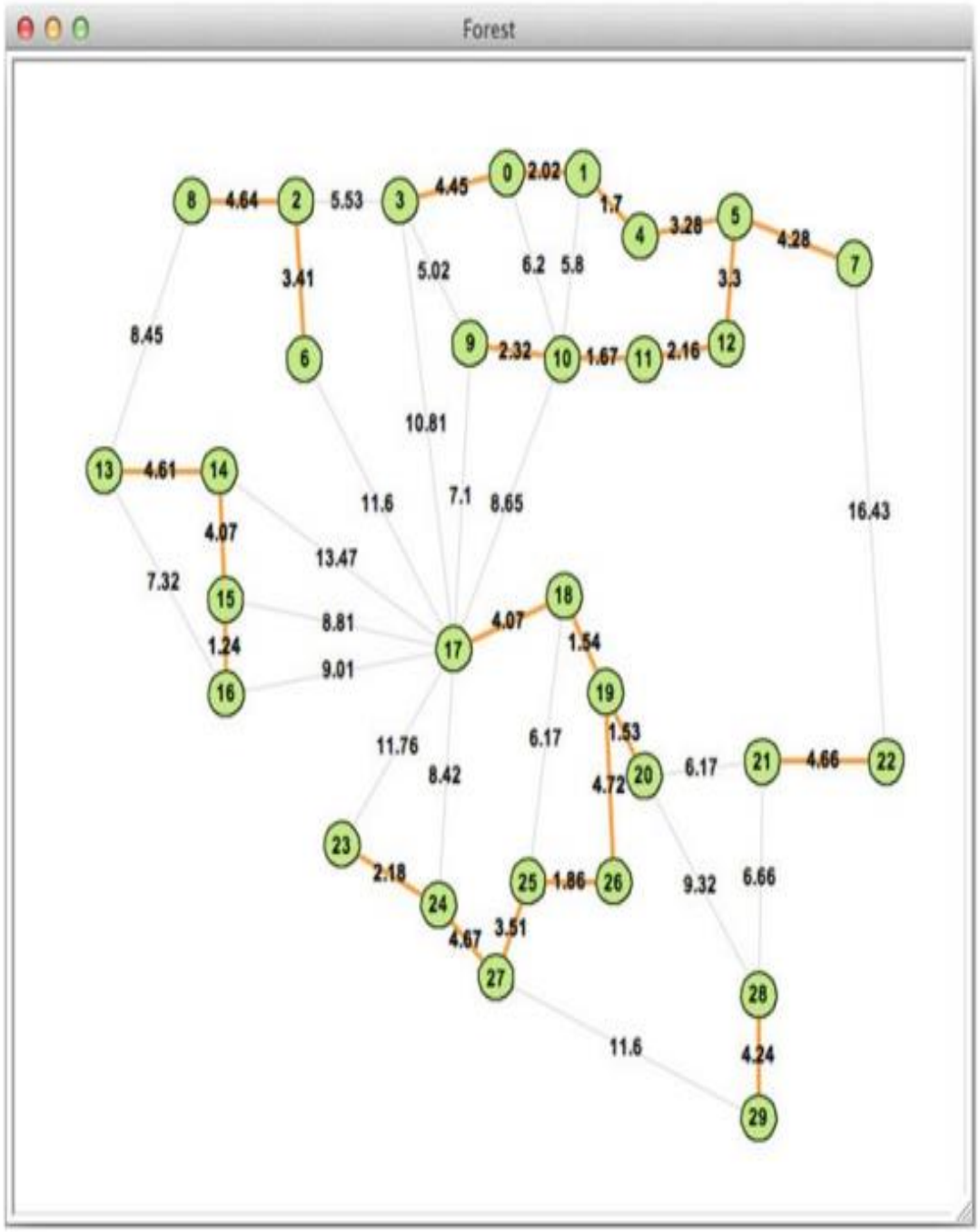
Şekil 7.5 Minimum Ağırlıklı Yayılan Ağaç



Ağaçlar, çizge kuramı bağlamında, tüm olası çizgeler kümesinin bir alt kümesidir. Ağaç, herhangi bir döngüsü olmayan bir çizgedir. Buna ek olarak, bir ağacın köşe sayısından bir daha az kenar içermesi gerektiğini kanıtlamak nispeten kolaydır. Aksi takdirde, bir ağaç olmazdı

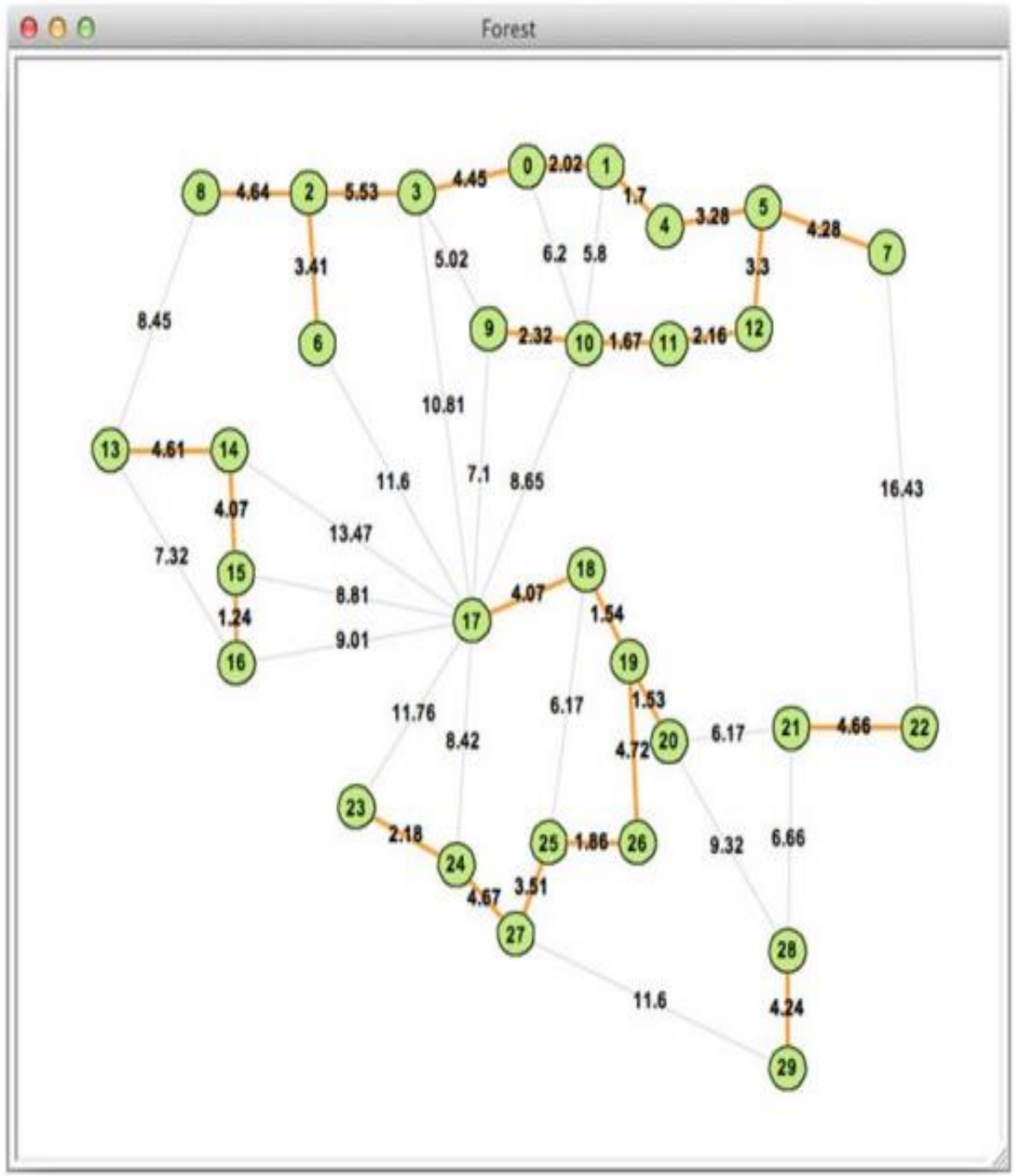
- Önceki şekilde grafik için ağaç kenarları turuncu ile vurgulanmış bir minimum ağırlıklı yayılan ağaç içermektedir. Minimum ağırlıklı yayılan ağaç demiyoruz çünkü genel olarak birden fazla minimum ağırlıklı yayılan ağaç olabilir. Bu durumda, muhtemelen yalnızca bir tane mümkündür.

- Kruskal'ın algoritması açgözlü bir algoritmadır. Açgözlü terimi, algoritmanın bir alternatifler listesi sunulduğunda her zaman ilk alternatifi seçtiği ve seçim yaparken asla hata veya yanlış seçim yapmadığı anlamına gelir. Başka bir deyişle, Kruskal'ın algoritmasında geri izleme gerekmez.



Şekil 7.6 Kruskal's: Anlık Görüntü 1

Algoritma, tüm kenarları ağırlıklarına göre artan sırada sıralayarak başlar. Grafiğin tamamen bağlı olduğu varsayıldığında, yayılan ağaç |V |-1 kenar içerecektir. Algoritma, grafikteki tüm köşelerin kümelerini oluşturur, her bir köşe için bir küme tepe noktası, başlangıçta sadece kümeye karşılık gelen tepe noktasını içerir. Şekil 7.3'teki örnekte, başlangıçta her biri bir tepe noktası içeren 30 küme vardır. Algoritma, yayılan ağaç kenarları kümesine |V |-1 kenar eklenene kadar aşağıdaki şekilde ilerler. 1. Bir sonraki en kısa kenar incelenir. Kenarın iki tepe noktası farklı kümelerdeyse, kenar güvenli bir şekilde yayılan ağaç kenarları kümesine eklenebilir. İki köşe kümesinin birleşiminden yeni bir küme oluşturulur ve önceki iki küme köşe kümeleri listesinden çıkarılır. 2. Kenarın iki tepe noktası zaten aynı kümedeyse, kenar yok sayılır. Algoritmanın tamamı bu. Algoritma açgözlüdür çünkü bir döngü oluşturmadığı sürece her zaman bir sonraki en küçük kenarı seçer. Bir kenar eklenerek bir döngü oluşturulacaksa, herhangi bir hatayı geri almak veya geriye gitmek zorunda kalmadan hemen bilinir. Şekil 7.6'yı düşünün. Bu anlık görüntüde algoritma zaten bir ağaç ormanı oluşturmuştur, ancak henüz bir yayılan ağaç oluşturmamıştır. Turuncu renkteki kenarlar yayılan ağacın bir parçasıdır.



Şekil 7.7 Kruskal's: Anlık Görüntü 2

Bir sonraki en kısa kenar olan 3. köşeden 9. köşeye giden kenar şu anda değerlendirilmektedir. 3 ve 9'u içeren küme {3, 0, 1, 4, 5, 12, 11, 10, 9} kümesidir. Uç noktaları 3 ve 9 olan kenarın eklenmesi yapılamaz çünkü 3 ve 9 köşeleri zaten aynı kümededir. Dolayısıyla, bu kenar atlanır. Minimum ağırlıklı yayılan ağacın bir parçası olamaz. Bir sonraki en kısa kenar 2 ve 3 köşeleri arasındaki kenardır. 2, {8, 2, 6} kümesinin bir üyesi ve 3 de bir önceki paragraftaki kümesinin bir üyesi olduğundan, {2, 3} kenarı minimum ağırlıklı yayılan ağaç kenarlarına eklenir ve yeni {8, 2, 6, 3, 0, 1, 4, 5, 12, 11, 10, 9} kümesi önceki iki alt kümesinin yerini alarak aşağıdaki şekilde gösterildiği gibi oluşur Şekil 7.7. Bir sonraki en kısa kenar 1 ve 10 köşeleri arasındaki kenardır. Bu kenar yine eklenemez çünkü 1 ve 10 aynı kümededir ve bu nedenle kenarın eklenmesi bir döngü oluşturacaktır. Bir sonraki en kısa kenar 18 ve 25 köşeleri arasındaki kenardır, ancak yine eklenmesi bir döngü oluşturacağından atlanır. Algoritma şu şekilde ilerler V |-1 kenarlı bir yayılan ağaç oluşana kadar bu şekilde devam eder (grafiğin tamamen bağlı olduğu varsayılır).

7.4.1 Doğruluk Kanıtı

•Kruskal'ın algoritması, minimum ağırlıklı yayılan ağacı (MST) bulmak için kullanılır. Bu algoritma, açgözlü bir yaklaşım kullanır ve her adımda en küçük ağırlıklı kenarı seçer. Algoritmanın doğruluğunu kanıtlamak için, çelişki yoluyla bir ispat kullanılır.

- İspat şu adımları içerir:
- Algoritma tarafından bulunan ağacın bir MST olmadığını varsayalım.
- Temel bilgiye dayanarak, bir ağaçta |V| - 1 kenar olması gerektiğini kabul edelim.
- Başka bir ağaç (T') Kruskal'ın bulduğu ağaçtan daha küçük toplam ağırlığa sahip olur.
- T' ağacında T'den farklı bir kenar olduğunu varsayalım.
- Bu yeni kenar sadece bir döngü oluşturabilir.
- Bu durumda, döngüdeki yeni kenar, eski ağacın bir kenarını siler ve bu silinen kenarın ağırlığı, yeni eklenen kenardan daha fazla olmalıdır.
- Eğer yeni eklenen kenarın ağırlığı, silinen eski kenarın ağırlığıyla aynı ise, Kruskal'ın algoritması zaten doğru bir çözüm bulmuş olur.
- Bu çelişki, Kruskal'ın algoritmasının her zaman minimum ağırlıklı yayılan ağacı doğru bir şekilde bulduğunu gösterir.

7.4.2 Kruskal'ın Karmaşıklık Analizi

- Kruskal'ın algoritmasının karmaşıklığı, kenar listesini sıralamaya ve ardından algoritma ilerledikçe kümelerin birleşimini oluşturmaya bağlıdır. Bölüm 4'te quicksort'un karmaşıklığını incelerken gösterildiği gibi bir listeyi sıralamak O(|E|log|E|)'dir.
- Listeyi sıralamak Kruskal'ın algoritmasının bir yarısıdır. Diğer yarısı ise doğru kenarları seçmektir. Her kenarın kendi başına bir kümede başladığını ve iki uç nokta ayrı kümelerdeyse bir kenarın minimum ağırlıklı yayılan ağaca ait olduğunu hatırlayın. Eğer öyleyse, uç noktaları içeren iki küme için bir birleşme oluşturulur ve bu iki kümenin birleşmesi ileriye doğru önceki iki kümenin yerini alır
- Algoritmanın bu bölümünü uygulamak için üç işlem gereklidir.
- 1. Öncelikle, yayılan ağaca eklenmesi düşünülen kenarın her bir uç noktası için kümeyi keşfetmeliyiz.
- 2. Daha sonra iki set eşitlik açısından karşılaştırılmalıdır.
- 3. Son olarak, iki kümenin birleşimi oluşturulmalı ve gerekli güncellemeler yapılmalıdır, böylece iki uç nokta köşesi artık orijinal kümeleri yerine iki kümenin birleşimini ifade eder

•Bu bölümde, Kruskal'ın algoritmasını uygularken kümeler listesi kullanılarak verimliliğin nasıl artırılabileceği açıklanmaktadır.

- **Kümeler Listesi Oluşturma**: Listedeki her konum, grafikteki bir köşeye karşılık gelir. Köşeler 0'dan başlayan tamsayı tanımlayıcılar olabilir.
- **Sabit Zamanlı İşlemler**: Bir listede indeksli arama O(1) zamanda yapılabilir, bu sayede bir köşeye karşılık gelen küme O(1) zamanda belirlenebilir. Aynı şekilde, iki kümenin aynı olup olmadığını O(1) zamanda belirlemek mümkündür. Python'da `is` anahtar sözcüğü bu işlem için kullanılır.
- **Yeni Küme Oluşturma**: Yeni bir küme oluşturmak, yani iki kümeyi birleştirmek, en kötü durumda O(|V|^2) karmaşıklığına sahiptir. Bu işlem |V| - 1 kez gerçekleştirilir. İlk kez gerçekleştiğinde mevcut kümeye bir köşe eklenir, sonraki seferlerde daha fazla köşe eklenir. Bu, en pahalı işlemdir.
- Sonuç olarak, Kruskal'ın algoritmasının verimliliğini artırmak için, sabit zamanlı kümeler işlemlerini kullanmak ve yeni küme oluşturma işlemini optimize etmek önemlidir. Bir sonraki bölümde, bu durumu önemli ölçüde iyileştiren bir veri yapısı tanıtılmaktadır.

7.4.3 Bölme Veri Yapısı

Üçüncü gerekli işlem olan iki kümenin tek bir kümede birleştirilmesini geliştirmek için, Partition adı verilen özel bir veri yapısı kullanılabilir. Partition veri yapısı, her tepe noktası için bir girdisi olan bir tamsayı listesi içerir. Başlangıçta, liste basitçe indisleriyle eşleşen tamsayıların bir listesini 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, şimdiye kadar inşa edilen yayılan ormandaki bağlantılı kenar kümelerini temsil eden bir ağaç listesi olarak düşünün. Bir ağacın kökü, listedeki bir konumdaki değer indeksiyle eşleştiğinde gösterilir. Başlangıçta, bölüm içindeki her tepe noktası kendi kümesindedir çünkü her tepe noktası kendi ağacının köküdür

- Bir tepe noktası için kümeyi keşfetmek, ağacı köküne kadar izlemek anlamına gelir. Şekil 7.6'da gösterildiği gibi 3. vertex'ten 9. vertex'e giden kenarın minimum ağırlıklı yayılan ağaca eklenmesi düşünüldüğünde ne olacağını düşünün. O zamanki bölüm şu şekilde görünür.

- 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]

- Vertex 3 şu anda kendi ağacının kökü değildir. 7 indeksi 3'te bulunduğundan, daha sonra bölümleme listesinin 7 indeksine bakarız. Bölümleme listesindeki bu konum değeriyle eşleşir. 3, 7 konumunda köklenmiş kümede (yani ağaçta) bulunmaktadır. Daha sonra tepe noktası 9'a baktığımızda, listedeki 9. indeks 11'i içerir. Listedeki 11. dizin 7'yi içerir. Vertex 9 da index 7'de köklenen kümede (yani ağaçta) yer alır. Dolayısıyla, 3 ve 9 numaralı tepe noktaları zaten aynı kümededir ve bir döngü oluşacağı için 3'ten 9'a giden kenar minimum yayılan ağaca eklenemez. Dikkate alınacak bir sonraki kenar 2 ve 3 köşeleri arasındaki kenardır. 2'yi içeren ağacın kökü bölümün 2. indeksindedir. Az önce gördüğümüz gibi 3'ü içeren köşenin kökü 7. indekstedir. Bu iki köşe aynı kümede değildir, bu nedenle 2'den 3'e olan kenar minimum yayılan ağaç kenarlarına eklenir.

- Gerçekleştirilmesi gereken üçüncü işlem, 2 ve 3'ü içeren kümelerin birleştirilmesidir. İşte bu noktada bölümleme işe yarar. İki ağacın kökünü bulduktan sonra, ağaçlardan birinin kökünü diğer ağacın köküne işaret ettiririz. Bu iki kümeyi birleştirdikten sonra bu bölümlemeyi elde ederiz.

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

- Algoritmanın bu noktasında, 7 köklü ağaç 2 köklü olacak şekilde değiştirilmiştir. Tepe noktası 2 ve 3'ü içeren iki kümeyi birleştirmek için gereken tek şey buydu! İki küme birleştirildiğinde bir ağacın kökü diğer ağacın kökünü gösterecek hale getirilebilir.

•Bu bölümde, Kruskal'ın algoritmasında kullanılan bir veri yapısı olan "Bölme veri yapısı" ele alınmıştır. Bu yapı, aynı kümeye ait olup olmadığını kontrol etmek ve iki kümeyi birleştirmek için kullanılır.

SameSetAndUnion Yöntemi:

- İki tepe noktası numarası alır.
- İki köşe aynı kümeye sahipse `true` döner, değilse birleştirip `false` döner.

- **Kök Bulma**:
  - İki köşenin köklerini bulmak en kötü durumda O(|V|) zaman alır, ancak pratikte bu işlem çok daha hızlıdır.
  - Ortalama derinlik örneklerde 1.7428 ve 7.5656 olarak verilmiştir.
- **Ortalama Karmaşıklık**:
  - SameSetAndUnion yöntemi ortalama O(log|V|) karmaşıklığa sahiptir.
  - Kruskal'ın algoritmasının ortalama karmaşıklığı O(|E|log|V|) olur.

- **Genel Karmaşıklık**:
- Kenarları sıralamak O(|E|log|E|) zaman alır.
- Kruskal'ın algoritmasının genel ortalama karmaşıklığı O(|E|log|E|)'dir.

- **Verimlilik**:
- Kruskal'ın algoritması büyük ve çok kenarlı grafikler için bile minimum ağırlıklı yayılan ağacı hızlı bir şekilde bulur.
- Bu özet, Kruskal'ın algoritmasının verimliliğini ve bölme veri yapısının katkısını vurgular.

- Soru 3:Derinlemesine ilk aramada, ziyaret edilen setin amacı nedir?

•Derinlemesine İlk Arama (DFS, Depth-First Search) algoritmasında, ziyaret edilen bir küme (visited set) kullanmanın birkaç önemli amacı vardır:

- **Döngülerin Önlenmesi**:
  - Grafikte döngüler olabilir. Ziyaret edilen bir köşe kümesini tutarak, DFS'nin aynı köşeyi tekrar ziyaret etmesini engelleriz. Bu, algoritmanın sonsuz döngülere girmesini önler.
- **Tekrar Ziyaretlerin Önlenmesi**:
  - Ziyaret edilen köşeleri kaydederek, aynı köşeye birden fazla kez gitmeyi önleriz. Bu, gereksiz hesaplamalardan ve performans kaybından kaçınmamıza yardımcı olur.
- **Grafik Bileşenlerinin Tespiti**:
  - Ziyaret edilen küme, DFS'nin hangi köşelere ulaştığını ve hangi bileşenlerin bağlı olduğunu izlememize yardımcı olur. Bu, bir grafikte bağlı bileşenleri veya ayrık bileşenleri tanımlamak için önemlidir.
- **Doğru Yolun Takibi**:
  - Ziyaret edilen köşeleri izlemek, arama sırasında doğru yolun takip edilmesini sağlar. Bu, özellikle bir hedef düğüme ulaşmak veya belirli bir yapıyı aramak için önemlidir.

- Soru 4:Bir grafın derinlemesine ilk aramasında geri izleme nasıl gerçekleştirilir? Geri izlemenin nasıl gerçekleştiğini açıklayın.

•DFS, genellikle bir yığın (stack) kullanılarak uygulanır ve bu yığın, geri izlemeyi doğal olarak gerçekleştirir. DFS, bir düğümden başlayarak, önce mümkün olduğu kadar derine iner ve çıkmaz bir yola ulaştığında geri izleme yaparak bir önceki düğüme döner ve diğer komşuları ziyaret eder.

- **Adım Adım Açıklama**
- **Başlangıç Noktası**:
  - Bir başlangıç düğümü seçilir ve ziyaret edilir.
  - Bu düğüm yığına eklenir.
- **İlerle ve Ziyaret Et**:
  - Yığının tepesindeki düğümün ziyaret edilmemiş bir komşusu bulunur.
  - Bu komşu ziyaret edilir ve yığına eklenir.
-

•Soru 4 devamı:

•**Çıkmaz Yol**:

- Bir düğümün tüm komşuları ziyaret edilmişse veya hiç komşusu yoksa, çıkmaz yola ulaşılmış demektir.
- Bu durumda, yığının tepesindeki düğüm yığından çıkarılır (geri izleme yapılır).

- **Geri İzleme**:
  - Geri izleme, yığının tepesindeki düğümün bir önceki düğüme dönmesiyle gerçekleştirilir.
  - Bir önceki düğümün diğer komşuları ziyaret edilmemişse, bu komşular ziyaret edilir.
  - Eğer yığında daha ziyaret edilmemiş komşular kalmamışsa, süreç devam eder.
- **Tamamlanma**:
  - Yığın boşalana kadar süreç devam eder.
  - Tüm düğümler ziyaret edilene kadar döngü tekrarlanır.

Teşekkürler

Emin Batuhan PINAR

2023481031