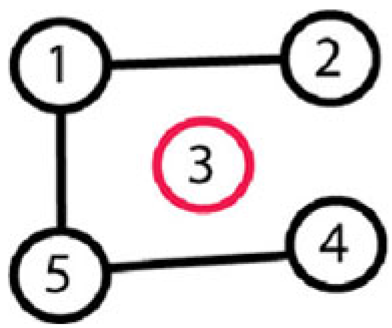
Ayrık Kümeler(Disjoint-Set Data Structure)(DSU)

Genellikle bir algoritmanın verimliliği kullandığımız veri yapılarına bağlıdır. Kullanacağınız veri yapısının doğru seçimi çalışma zamanından bellek kullanımına kadar bir çok alanda fayda sağlayabilir.

Problem

Bir odada N tane insan var. Herhangi ikisi doğrudan ya da dolaylı olarak arkadaşsa biz o ikisini arkadaş sayıyoruz. Yani A ile B arkadaş ve B ile C arkadaş ise A ile C de arkadaştır. İçindeki herhangi iki kişinin arkadaş olduğu gruplara arkadaş grubu diyelim. Size doğrudan arkadaş olan kişilerin listesi verilecek, Kaç farklı arkadaş grubu olduğunu bulunuz. Örneğin N=5 ve doğrudan arkadaşlık ilişkileri 1-2, 5-4, 5-1 olsun. Bu durumda 1-2-4-5 birbirleriyle arkadaş olacaklar 3'ün ise hiç arkadaşı olmayacak. Toplamda 2 farklı arkadaş grubu oluşmuş olacaktır.



Çözüm

Bu problem BFS ya da diğer graph algoritmalarıyla çözülebilir. Fakat biz DSU kullanarak çözeceğiz. DSU dinamik bir kümeler dizisi içerir(S_1,S_2,S_3...S_N). İki kümenin hiç ortak elemanı yoksa o iki küme ayrıktır. DSU yapısında her kümenin bir **temsilci**si vadır. Her kümenin kendi temsilcisi kendi içinden bir elemandır.

Bu soru için arkadaş grupları kümelerimiz olacak. Her grubun temsilcisi ise o gruptaki en büyük numaralı kişi olacak. İlk başta 5 kümemiz var: {1},{2},{3},{4},{5}. Yani herkes tek başına bir grup ve herkes kendi grubunun temsilcisi.

Sonraki adımda arkadaşlık bilgilerini gruplara işleyeğiz. 1 ile 2'yi arkadaş yapmak için iki kümeyi birleştirip yeni temsilciyi seçmek gerekiyor bu durumda yeni kümelerimiz şöyle olacak: {1,2},{3},{4},{5} ve {1,2} kümesinin temsilcisi 2 numaralı kişi olacak. Sonra aynı şekilde 4 ve 5'i arkadaş yapacağız. Kümeler {1,2},{3},{4,5} olacak ve {4,5} grubunun temsilcisi 5 numaralı kişi olacak. Son olarak 1 ile 5'i arkadaş yapacağız ve kümelerin yeni durumu {1,2,4,5},{3} olacak. Büyük kümenin temsilcisi ise 5 numaralı kişi diğerininki ise 3 numaralı kişidir(temsilcilere neden ihtiyaç duyduğumuzu sonra açıklayacağız). Bütün arkadaşlık ilişkilerini kümelere işledik ve şu an elimizde 2 küme var yani toplamda 2 farklı arkdaş grubu var.

Belki herhangi iki kişinin aynı kümede olup olmadığını nasıl kontrol edebileceğimizi merak ediyorsunuzdur. İşte temsilcileri bu aşamada kullanacağız. Örneğin 2 ve 3 numaralı kişilerin aynı grupta olup olmadığını merak ediyor olalım. Bunun için 2'nin olduğu kümenin temsilcisiyle 3'ün olduğu kümenin temsilcisinin aynı olup olmadığına bakmamız yeterli olacaktır. 2'nin kümesinin temsilcisi 5, 3'ünki ise 3'tür, yani 2 ve 3 farklı kümelerde bulunuyorlar.

Bazı İşlemler

create_set(x) --- sadece x elemanını içeren yeni bir küme oluşturur.
merge_sets(x,y) --- x ve y(bu ikisi farklı kümelerdeler)nin bulunduğu kümeleri birleştirir. x ve y'nin eski kümelerini yok eder.
find_set(x) --- x'in bulunduğu kümenin temsilcisini bulur.

Bu İşlemlerin Kullanıldığı Çözüm

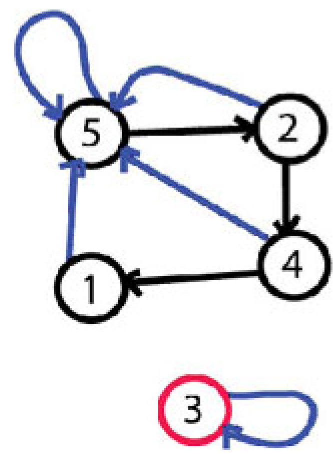
```
Oku -> N;
for (1'den N'e kadar bütün x'ler için)
        create_set(x)
for (Bütün (x y) arkadaşlıkları için )
        if (find_set(x) != find_set(y))
                merge_sets(x, y)
```

Eğer herhangi bir (x, y) ikilisinin aynı grupta olup olmadığını kontrol etmek istiyorsanız if (find_set(x) == find_set(y)) kontrolünü kullanabilirsiniz.

DSU'nun çalışma zamanını N ve M'e göre inceleyelim. N create_set(x) fonksiyonunu çağrılma sayısı, M ise 3 fonksiyonun toplam çağrılma sayısıdır. merge_sets(x,y) fonksiyonu her çağrılışında iki kümeyi yok edip bir küme oluşturuyor, yani toplam küme sayısını bir azaltıyor. Eğer başta K tane küme varsa merge_sets(x,y) fonksyonunu K-1 kere çağırarak küme sayısını 1'e indirebiliriz. Bu yüzden merge_sets(x,y) fonksiyonunu çağrılma sayısı create_set(x) fonksiyonunun çağrılma sayısından küçük eşit olacaktır.

Linked List(Bağlı Liste) İle Kodlama

DSU'yu linked list ile kodlayabiliriz. Her küme ayrı bir liste olacak. Her elemanın iki pointerı olacak bunlardan biri listedeki kümenin diğer elemanını diğeri ise kümenin temsilcisini işaret edecek. Aşağıdaki DSU'da linked list kullanımının resimli gösterimi mevcuttur. Mavi oklar temsilcileri, siyah oklar ise sonraki elemnaları gösteren pointerlardır. Bu şekilde create_set(x) ve find_set(x) fonksiyonlarının zaman karmaşıklıkları O(1) olmuş oldu. crete_set(x) fonsiyonu içinde sadece x olan bir linked list oluşturuyor. find_set(x) x'in kümesinin temsilcisini buluyor.



merge_sets(x,y) için kolay olan yol x'in bulunduğu listeyi y'nin bulunduğu listenin sonuna iliştirmek. Yeni kümenin temsilcisi y'nin kümesinin temsilcisi olur. Fakat bu durumda x'in eski kümesindeki her elemanın temsilci pointerını güncellememiz gerekir ki bu da kümedeki eleman sayısına bağlı olarak oldukça yavaş bir işlem olabilir. En kötü durumda zaman karmaşıkığı $O(M^2)$ olacak(M, merge_sets(x,y) fonksiyonunun çağrılma sayısı). Bu şekilde zaman karmaşıklığı fonksiyonun her çağırılışı için O(N) olmuş olur(N, bütün kümelerdeki elemanların sayısı).

Kümelere Ağırlıkları Dahil Etme

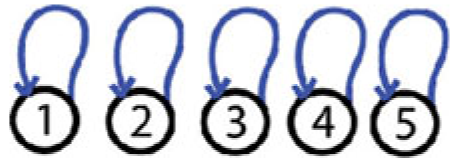
Bu durumda temsilci nodelar kendi kümeinde kaç eleman olduğunu da tutacaklar. Bu optimizasyonun amacı merge_sets(x,y) fonksiyonunun her zaman küçük olan kümeyi büyük olana bağlamasını sağlamak(eşitlik durumunda rastgele çalışması sıkıntı olmaz). Bu optimizasyon zaman karmaşıklığını O(N+MlogN)'e düşürür(M, 3 fonsiyonun toplam çağrılma sayılarıdır; N, create_set(x) fonksiyonunun çağrılma sayısıdır).

Şu ana kadar zaman karmaşıklığını O(M+NlogN)'e, bellek kullanımını ise O(N)'e getirebildik(N, kişi sayısı; M, doğrudan arkadaşlık ilişkisi sayısı). Yeni bir optimizasyon daha yapmamız gerekiyor.

Ağaçlar

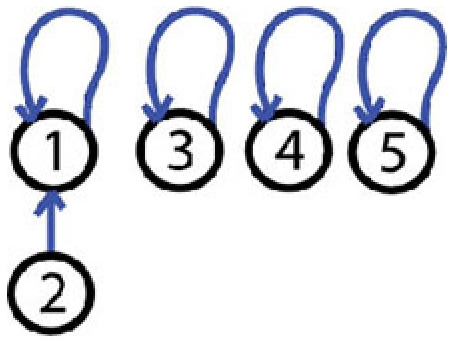
DSU'yu ağaçlar yardımıyla da kodlayabiliriz. Her küme bir ağaç olacak. Her node bir elemanı, her ağaç da bir kümeyi belirtecek. Her node ilk atasını işaret edecek, root nodeları ise kendilerini işaret edecekler. Yani kümelerin temsilcileri root nodeları olacak. Yukarıdaki problem için ağaçların nasıl görüneceğini adım adım inceleyelim.

Adım 1: Hiç bir arkadaşlık ilişkisi yapıya işlenmemişken



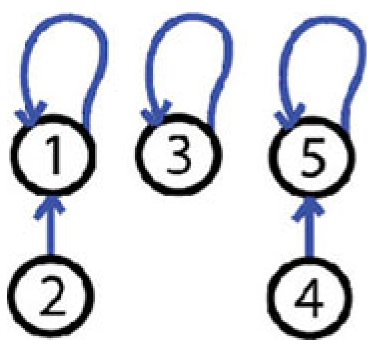
5 tane ağacımız var e her ağaçta 1 eleman var. Burada her eleman root(yani hepsi temsilci).

Adım 2: 1 ve 2 arkadaş oluyorlar --- merge_sets(1,2)

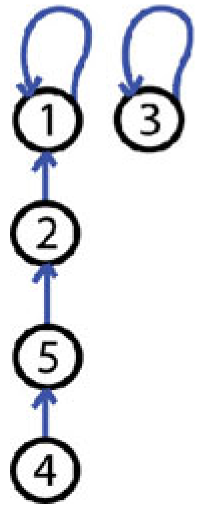


2’yi 1’e bağladık. 1 ile 2’nin bulunduğu ağacın temsilcisi 1.

Adım 3: 4 ve 5 arkadaş oluyorlar --- merge_sets(4,5)



Adım 4: 5 ve 1 arkadaş oluyorlar --- merge_sets(5,1)



Şu ana kadar ağaçlı yöntemin hız olarak linked listli olandan bir farkı yok.

İki Geliştirme Daha

“Union by rank” ve **“path compression”** geliştirmelerini kullanacağız. **“Union by rank”**dan başlayalım. rank[] dizisi her node için o nodeun alt ağacındaki en büyük derinliği tutuyor. Bu geliştirmedeki amacımız merge_sets(x,y) fonksiyonunda rootunun rank[] dizisisindeki değeri küçük olan ağacı diğerine bağlamak. Bu şekilde her nodeun rank[] dizisindeki değerinin üst sınırını o nodeun alt ağacındaki node sayısının 2 tabanına göre logaritması olarak belirlemiş oluyoruz. **“path compression”**daki amaç ise nodeların P[] dizisinde ilk atalarını değil de doğrudan içinde bulundukları ağacın rootunu işaret etmelerini sağlamak.

Bu optimizasyonları uygulamak için rank[] dizisi üzerine eğileceğiz. rank[x] x’in altındaki nodelarddan kendisine en uzak olan node ile arasındaki mesafeyi tutuyor(edge sayısı cinsinden). create_set(x) fonksiyonunda rank[x] değeri ilk başta 0 olarak atanmalıdır. merge_sets(x,y) fonksiyonunda rootunun rank[] dizisindeki değeri daha küçük olan ağacı diğerine bağlayacağız. Eşitlik durumunda ise istediğimiz ağacı diğerine bağlayabiliriz fakat bu durumda diğer ağacın rootunun rank[] dizisindeki değeri 1 artmış olacak.

P[x], x nodeunun atasını tutsun.

```
create_set(x)
        P[x] = x
        rank[x] = 0

merge_sets(x, y)
        PX =find_set(x)
        PY =find_set(y)
        If (rank[PX] > rank[PY])
```

```
        P[PY] = PX
    Else
        P[PX] = PY
    If (rank[PX] == rank[PY])
        rank[PY] = rank[PY] + 1
```

```
find_set(x)
    If (x != P[x])
        p[x] = FIND-SET(P[X])
    Return P[X]
```

Şimdi bu optimizasyonların bize çalışma zamanında ne kadar katkı sağladıklarını inceleyelim. Eğer sadece “**union by rank**”i kullansaydık linked listli olandan pek bir farkı olmayacaktı. Ama iki optimizasyonu da beraber kulandığımızda zaman karmaşıklığı $O(m\,\alpha(m,n))$ olur. $\alpha(m,n)$ fonksiyonu Ackermann fonksiyonunun ters fonksiyonudur, yani m ve n arttıkça fonksiyonun değeri çok çok yavaş artar hatta çoğu uygulamada 4’ten küçük eşit olarak kalır.

Probleme Dönüş

İlk başta anlattığımız problem DSU ile O(N + M) zaman karaşıklığında ve O(N) bellek kullanımında çözülebilir. Çalışma zamanı BFS’li çözümden çok da farklı olmadı ama bellekten epey tasarruf etmiş olduk. Problemi şu şekilde değiştirdiğimizi düşünün: Odada N kişi var ve Q tane işlem var; “x y 1” şeklindeki işlem x ile y’nin artık arkadaş olduklarını belirtir, “x y 2” şeklindeki ise x ile y’nin o an aynı arkadaş grubunda olup olmadığını bulmamızı ister. Bu durumda DSU’yu kullanmak çok daha avantajlı olacaktır(zaman karmaşıklığı O(N+Q) olur).

DSU Kullanarak Çözülebilecek Problemler

http://community.topcoder.com/stat?c=problem_statement&pm=2998
http://community.topcoder.com/stat?c=problem_statement&pm=7921
http://community.topcoder.com/stat?c=problem_statement&pm=1110
http://community.topcoder.com/stat?c=problem_statement&pm=2932
http://www.iarcs.org.in/inoi/online-study-material/problems/roads.php
http://www.link.cs.cmu.edu/contest/130220/problems/B.pdf

Kaynaklar ve İlgili Linkler

http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=disjointDataStructure
http://en.wikipedia.org/wiki/Disjoint-set_data_structure
http://en.wikipedia.org/wiki/Ackermann_function