Ara Sınav 2 Çözümleri

Problem 1. ***Statik Grafik Gösterimi***

$G = (V, E)$ yönlendirilmemiş seyrek bir grafik olsun. $V = \{1, 2, \ldots, n\}$ olsun. $v \in V$ ve $i = 1, 2, \ldots,$ *out-degree(v)* için, *v*'nin ***i'ninci komşusunu***, *i*'ninci en küçük *u* köşesi olarak tanımlayın. Öyle ki, $(v, u) \in E$ olduğunda, v'ye komşu olan köşeleri sıralarsak; *u, i*'inci küçük köşe olsun.

Aşağıdaki sorguları sağlayan G grafiğinin bir gösterimini hazırlayın.

- DEGREE(v): v köşesinin derecesini döndürür.
- LINKED(u, v): Eğer bir kenar, *u* ve *v* köşesini bağlıyorsa; DOĞRU döner, aksi halde YANLIŞ döner.
- NEIGHBOR(v, **i**): *v*'nin *i*'inci komşusunu döndürür.

Veri yapınız asimptotik olarak, olabildiğince az yer kaplamalıdır. İşlemler de asimptotik olarak olabildiğince hızlı çalışmalıdır; ancak yer süreden daha önemlidir. Veri yapınızı yer ve süre bakımından analiz edin.

Çözüm: Veri Yapısı için, $\Theta(E)$ kadar yer ve her 3 işlem için $\Theta(1)$ süreye ihtiyaç duyan bir sonuç veriyoruz. Eğer $(u, v) \in E$ ise *(u, v)*'yi, eğer u'nun i'inci komşusu varsa *(u, i)*'yi tutan bir kıyım tablosu yaratın. Bazı *u*'lar ve *v* = *i*'ler için, hem *u* ile *v*'nin komşu olması hem de *u*'nun *i*'ninci komşusu olması mümkündür. Bu durum, kıyım tablosundaki her kayıtta uydu verileri tutularak çözülür. Kıyım tablosundaki (u,i) anahtarlı kayıt için, eğer u köşesinin, v=i olan bir komşusu varsa, kayıtta bu durumu belirten bir bit saklayın; eğer u bir i'ninci komşuya sahipse, ilgili komşu köşenin endeksini kayıtta saklayın. Ayrıca, başka bir köşeye bağlantısı olan her u köşesinin (u,1) anahtarı için derecesini kayıtta saklayın.

Böylece anahtarın birinci koordinatında her u köşesinin, kıyım tablosunda en fazla *degree(u) + degree(u)=2 degree(u)* girdisi vardır. Toplam girdilerin sayısı en fazla;

$$\sum_{u=1}^{n} 2\,\mathrm{degree}(u) = 4|E|$$

olur. *Mükemmel kıyımı* kullanarak ve veri yapısı hazırlanırken, az sayıda rastgele örneklem olacak şekilde, uygun bir kıyım fonksiyonu seçerek(veri yapısının kuruluşu evresinde), arama süresini $\Theta(1)$'e, alan gereksimini de depolanacak girdiler açısından doğrusal düzeye, mesela $\Theta(E)$'ye çekebiliriz(Kitabın 249. sayfasında Corollary 11.12'ye bakın). Aynı kıyım fonksiyonları ailesini, kitaptaki gibi, her 2 boyutlu *(u,v)'yi,* $[1....n^2]$ aralığında $(u-1)n + v$ farklı sayıya çevirmek için kullanabiliriz. Böylece derece dizilimi ve kıyım tablosu için gerekecek toplam alan $\Theta(V+E)$ olur.

Bu durumda, DEGREE*(v), (v,1)* anahtarı için kıyım tablosuna bakar. Eğer bulursa, kayıtta saklanan derece değerini döndürür. Aksi durumda, v herhangi bir köşeye komşu olmadığından 0 döndürür. Bu $\Theta(1)$ süre alır. LINKED*(u,v), (u,v)* anahtarı için kıyım tablosuna bakar ve eğer tabloda varsa DOĞRU döndürür ve kıyım tablosundaki ilgili bit ayarlanır. Bu $\Theta(1)$ süre alır. NEIGHBOR*(v,i), (v,i)* anahtarı için kıyım tablosuna bakar, eğer anahtar varsa ve komşu köşe kayıtta saklanmışsa, o köşenin anahtar listesini döndürür. Bu da $\Theta(1)$ süre alır.

Problem 2. ***Video Oyun Tasarımı***

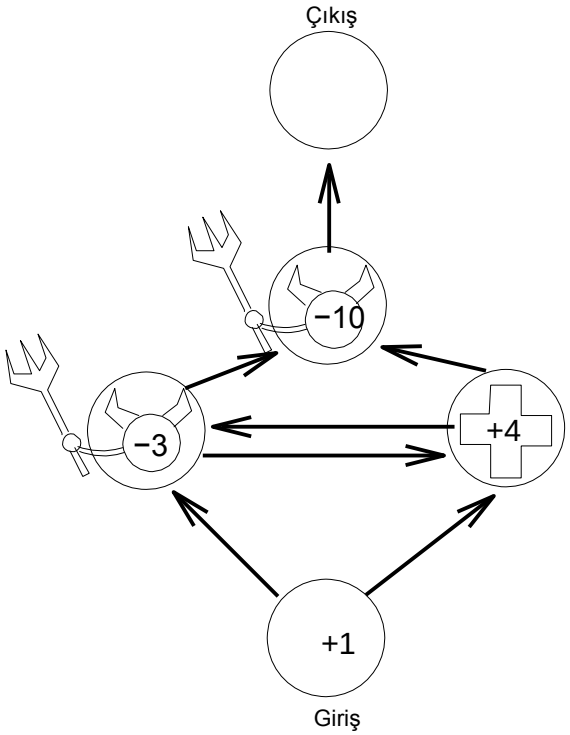
Profesör Cloud, uzun zamandır beklenen ***Take-home Fantasy*** isimli yılın oyununun tasarımına danışmanlık yapıyor. Oyundaki seviyelerden birinde, oyuncular girişten çıkışa gitmek için bir labirentteki birçok odayı gezmek zorunda. Odalarda hayat iksiri veya canavar olabileceği gibi, odalar boş da olabilir. Oyuncu odalarda dolaşırken *L* ***yaşam puanları*** azalır veya artar. Hayat iksiri içmek *L*'yi arttırırken, canavarla savaşmak *L*'yi azaltır. Eğer ***L***, 0'a veya altına düşerse oyuncu ölür.

Şekil 1'de gösterilen bu labirent; köşelerin odaları, tek yönlü kenarların da koridorları simgelediği bir *G=(V,E)* yönlendirilmiş grafiği ile simgelenebilir. Bir $\mathbf{f} : V \rightarrow Z$ köşe-ağırlık fonksiyonu odanın içindekileri simgeler.

- Eğer $\mathbf{f}(v) = 0$ ise oda boştur.
- Eğer $\mathbf{f}(v) > 0$ ise odada hayat iksiri vardır. Oyuncu odaya her girdiğinde *L* yaşam puanı f(v) kadar artar.
- Eğer $\mathbf{f}(v) < 0$ ise odada canavar vardır. Oyuncu odaya her girdiğinde, *L* yaşam puanı |f(v)| kadar azalır. L artı olmazsa oyuncu ölür.

Labirente ***giriş*** odası s Є V, ***çıkış*** odası da t Є V'dir. s'den her v Є V'ye ve her v Є V'den t'ye bir yol olduğunu kabul edin. Oyuncu, $L = L_0$ yaşam puanı ile girişte oyuna başlar, yani L_0 girişteki *f* değeridir. Şekilde $L_0 = 1$'dir.

Profesör Cloud canavarları ve hayat iksirlerini labirente rastgele yerleştirmek için bir program tasarlamış, ama bazı labirentlerde girişten çıkışa ulaşabilmek, $L_0 > 0$ değilse mümkün olmayabiliyor. s'den t'ye giden yolda, eğer oyuncu sağ kalabiliyorsa bu yol "güvenli"dir, yani yaşam-puanları hiç 0 olmuyordur.Oyuncu L_0=r ile başladığında,labirentte güvenli bir yol varsa, o labirenti **r-*girilebilir*** olarak tanımlayın.



Şekil 1: 1-girilebilir Labirent Örneği

Profesöre *r*'nin en küçük değerini belirlemek için verimli bir algoritma tasarlamasında yardımcı olun; öyle ki, bu algoritma, verilen labirentin *r-girilebilir* olduğunu veya böyle bir *r* olmadığını belirlesin. (Kısmi puan almak için, bir labirentin verilen bir *r* için *r*-girilebilir olup olmadığını belirleme problemini çözün.)

Çözüm:
O(V E lg r) sürede çalışan bir algoritma verebiliriz; burada *r* labirentin girilebilir olabilmesi için gereken en az hayat puanıdır. Bu soru 2 bölümde çözülür.

- Verilen bir *r* için labirentin girilebilir olup olmadığını anlayan bir algoritma: Bu Bellman-Ford'un değiştirilmiş sürümü ile yapılır. Her *u* düğümü için, oyuncunun *u*'ya eriştiğinde sahip olacağı en fazla puan *q[u]*'yu hesaplar. Oyuncu eksi puan ile *u*'ya ulaştığında öleceğinden, q[u]'nun değeri ya $-\infty$ (oyuncunun u'ya ulaşamayacağı anlamına gelir), ya da artıdır. Yani, eğer *q[t]* artı değerli ise (t çıkış düğümüdür), grafiğimiz r-girilebilir olur.

- En küçük *r*'nin 1'den küçük olamayacağını biliyoruz. Yani üstel ve ikili aramanın bir karışımını kullanarak *r*'nin en küçük değerini bulabiliriz. Bellman-Ford'un değiştirilmiş hali ile en küçük ***r***'yi, *log r* sürede bulabiliriz.

Algoritmanın çalışma süresi *O(V E log r)'dir; r,* en küçük *r*'dir ve labirent de *r-girilebilir*'dir.

Verilen bir *r* için girilebilirliğin anlaşılması : Bunun için Bellman-Ford'un değiştirilmiş bir sürümünü kullanırız. Verilen bir *r* için, her *u* düğümü için, oyuncunun u'ya erişmesi durumunda sahip olacağı en büyük *q[u]* puanı bulunur. Eğer, *q[t]* artı bir değer ise grafiğimiz r-girilebilirdir.

*u*Є*V* olan her köşe için, *q[u]*'nun alt sınırı olan *p[u]* korunur. Giriş hariç bütün *p[u]*'ları, başlangıçta – ∞ olacak şekilde ilklendiririz; sadece giriş *r* ile ilklendirilir. Bellman-Ford algoritmasını çalıştırıp, *p[u]* değeri artıp *q[u]*'ya yakınsayana kadar kenarları gevşetiriz (eğer pozitif ağırlık çevrimi yoksa). Burada, eksi değerli bir düğüme ulaşmanın, bu düğüme ulaşamamaktan daha iyi olmadığını fark etmek önemlidir. Yani, *p[u]* eğer artı oluyorsa onu değiştiririz, aksi takdirde, – ∞ olarak saklarız. *Relaxation*, yani Gevşetme ya da dengeye dönme yordamını aşağıdaki gibi değiştiririz.

```
V-RELAX(u, v)
1  if (u, v) ∈ E
2     then if ((p[v] < p[u] + f[v]) and (p[u] + f[v] > 0))
3     then p[v] ← p[u] + f[v]
4  π[v] ← u
```

Bütün kenarlar, *V* kez dengeye döndükten sonra, eğer eksi ağırlıklı hiçbir devre yoksa, bütün *p[u]*'lar karşılık gelen *q[u]*'ya yakınsamış olur (*u* köşesine erişme halinde sahip olunan en çok puan sayısı). Bu noktada, eğer *q[t]* artı değerli ise, oyuncu oraya artı hayat puanları ile ulaşabilir ve grafik r-girilebilir olur. Eğer, *p[t]* artı değerli değil ise, bütün kenarları bir kere daha dengeye döndürürüz (Tıpkı Bellman-Ford'da olduğu gibi). Eğer herhangi bir düğümün *p[u]*'su değişirse, *s*'den *r* puanla başlandığında ulaşılabilecek artı ağırlıklı bir çevrim bulduk demektir. Yani, oyuncu t'ye ulaşmak için gerekli olan puana erişebilmek amacıyla çevrimde yeterli kere dolaşabilir ve böylece grafik r- girilebilir'dir.. Eğer ulaşılabilecek bir artı ağırlık çevrimi bulamazsak, ve p[t] – ∞ ise, grafik r-girilebilir değildir. Algoritmanın doğruluğu Bellman-Ford'un doğruluğuna bağlıdır ve koşma süresi O(V E)'dir.

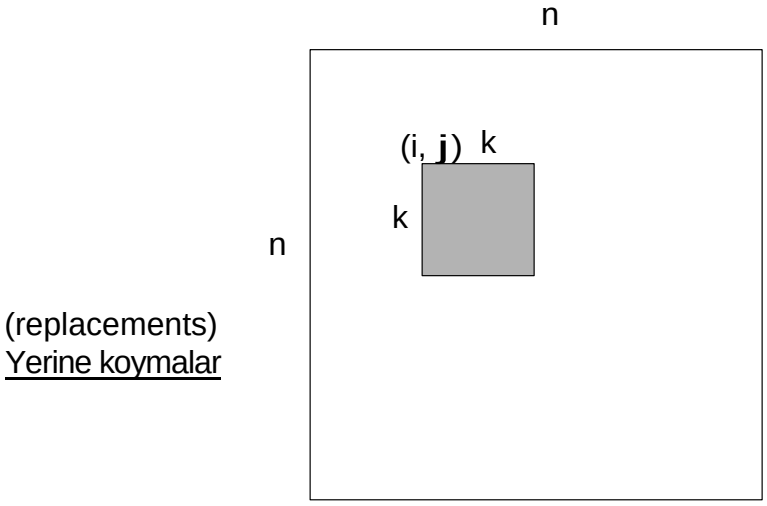
Grafiğin r-girilebililir olması için en küçük r'yi bulmak. Yukarıda verilen alt yöntemi kullanarak en küçük r'yi buluruz. Önce grafiğin 1-girilebilir olup olmadığını kontrol ederiz. Eğer öyle ise 1'i cevap olarak döndürürüz. Eğer değilse, 2- ve 4-girilebilirliği kontrol ederiz. i'inci basamakta 2^{i-1}'i kontrol ederiz. En sonunda 2^{k-1}'in girilebilir olmadığı ama 2^k'nın girilebilir olduğu bir k buluruz. Yani, *r*'nin en küçük değeri bu iki değer arasındadır. Bundan sonra $r=2^{k-1}$ ile $r=2^k$ arasında *r*'nin tam değerini bulmak için ikili arama yaparız.

Çözümleme : k = [lg r] olduğundan, yapılan işlemlerin sayısı *k + O(lg r) = O(lg r)*'dir. Yani, Bellman-Ford'u O(lg r) kere koşturmak zorunda kalırsınız ve toplam koşma süresi O (V E lgr) olur.

Alternatif Çözümler : Bazıları çıkıştan başlayarak, derinliğine veya enine arama ile düğümleri ziyaret edip, u'dan çıkışa ulaşmak için gereken en az puanı bulmak amacıyla kenarlarda dengeye dönme uygulaması yapmışlar. Bu tip bir yaklaşımla algoritma, artı ağırlıklı çevrimi olduğu takdirde, *O(M (V + E))* sürede koşar; burada M bütün canavarların toplam puanıdır. Bu sayı, gerçek *r* sayısı küçük olsa bile büyük olabilir. Bazıları, aynı şeyi arama yerine Bellman-Ford'u kullanarak yapmışlar. Bu da *O(M V E)* sürede koşar. Bunun yanısıra, $O(V^2E)$ sürede koşan birkaç akıllı çözüm daha vardı.

Problem 3. ***Görüntü Filtreleme***

İki-boyutlu filtreleme görüntü ve resim işlemede yaygın bir işlemdir. Bir resim, gerçek sayılardan oluşan *n x n* boyutlu bir matris ile simgelenir. Şekil 2'de görüleceği gibi, *k x k* boyutunda bir pencereyi matris boyunca dolaştırmak ve penceredeki her olası yerleşim için filtrenin penceredeki tüm değerlerin "çarpımını" alması buradaki fikirdir. "Çarpım" tipik bir çarpma işlemi değildir.



Şekil 2: Çıktı elemanı b_{ij} taranmış bölgedeki bütün a'ların çarpımıdır.

Bu problem için, işlemin birleşmeli ve değişimli özelliklere sahip bir ikili bir işlem olduğunu varsayabiliriz. Bu işlemde özdeşlik elemanı ***e*** için;

$$x \otimes e = e \otimes x = x$$

Örneğin, bu işlem, 0 özdeşlik elemanı ile toplama, 1 ile çarpma, ∞ ile **min** v.s. olabilir. Burada önemli olan, bu işlemin – ve + gibi tersinin doğru olduğunu varsaymamanız.

Daha açık olmak için, n x n bir resim verilmekte,

$$A = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \dots & a_{(n-1)(n-1)} \end{pmatrix}$$

(k × k)*-filtrelenmiş* görüntü n × n matrisidir.

$$B = \begin{pmatrix} b_{00} & b_{01} & \dots & b_{0(n-1)} \\ b_{10} & b_{11} & \dots & b_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{(n-1)0} & b_{(n-1)1} & \dots & b_{(n-1)(n-1)} \end{pmatrix}$$ i, j = 0,1,...,n-1 için,

$$b_{ij} = \bigotimes_{x=i}^{i+k-1} \bigotimes_{j=y}^{j+k-1} a_{xy}$$

(Kolaylık olsun diye, $x \geq n$ veya $y \geq n$ için $a_{xy} = e$ olduğunu varsayarız.)

A girdi matrisinin (k x k) - filtresini hesaplamak için verimli bir algoritma verin. Algoritmanızı çözümlerken, *k*'yı sabit bir değer olarak kabul etmeyin. Koşma sürenizi *n* ve *k* cinsinden ifade edin. (Kısmi puan almak için problemi bir boyutta çözün.)

Çözüm:

Bu 2 boyutlu filtreleme problemini $\Theta(n^2)$ sürede, onu önce n elemanlı iki adet tek boyutlu probleme indirgeyerek, sonra da tek boyutlu bir filtrelemeyi $\Theta(n)$ sürede nasıl çözebildiğimizi göstererek çözeriz. Filtrelenmiş $k > n$ değerleri, $k = n$ ile aynı olduğundan, $k \le n$ olduğunu varsayarız. Ortada, n^2 kadar hesaplanması gereken değer olduğundan, $\Theta(n^2)$ süreli algoritma en uygunudur. Aradaki C matrisini şu şekilde tanımlarız:

i, j= 0,1,...,n-1 için

$$C = \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0(n-1)} \\ c_{10} & c_{11} & \cdots & c_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(n-1)0} & c_{(n-1)1} & \cdots & c_{(n-1)(n-1)} \end{pmatrix}$$

$$c_{ij} = \bigotimes_{y=j}^{j+k-1} a_{iy}$$

C, *A*'nın her satırındaki tek boyutlu *k-filtresi*dir. Bu durumda,

$$\begin{aligned} b_{ij} &= \bigotimes_{x=i}^{i+k-1} \bigotimes_{y=j}^{j+k-1} a_{xy} \\ &= \bigotimes_{x=i}^{i+k-1} c_{xj} \ , \end{aligned}$$

olur, ve *B* de C'nin her sütunundaki tek boyutlu *k*-filtresidir.

Geriye kalan, 1 boyutlu k-filtrelemeyi hesaplamak için verimli bir metod tasarlamaktır. n uzunluğundaki bir dizilimde, tek boyutlu problemi çözmek için saf algoritma $\Theta(k\,n)$ zaman kullanır. Tek boyutlu algoritmayı bu 2 boyutlu problemin çözümünde kullanırsak, C'yi A'dan ve B'yi C'den hesaplanmak için gereken süre, iki hesaplama için de $\Theta(kn)$ gerektirir; bu da toplamda $\Theta(kn^2)$ demektir. Birçok öğrenci, tek boyutlu problemi $\Theta(n \lg k)$ sürede çözen, böylece 2 boyutlu problemi de $\Theta(n^2 \lg k)$'da çözen yollar bulmuşlar. Buna karşılık, tek boyutlu problemin Θ(n) sürede, böylece 2 boyutlu problemin de $\Theta(n^2)$ sürede çözüldüğünü keşfedenler olmuş.

Tek boyutlu problem için $\Theta(n)$ süreli algoritma aşağıdaki gibidir. Girdi dizilimimiz;

$$A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$$

ve k-filtreli çıktı dizilimimiz de;

$$B = \langle b_0, b_1, \ldots, b_{n-1} \rangle$$

olsun. Burada,

$$b_i = \bigotimes_{x=i}^{i+k-1} a_x$$

olur.

Genellemeyi bozmadan, *n*'nin *k*'ya tam olarak bölünebildiğini varsayalım. Aksi takdirde, *a*'nın sonunu *e* kimlik elemanları ile doldurarak, *n*'yi k'nın bir katı haline getirebilmek için n'yi ikiyle çarpmaktan daha çok şey yapmamız gerekir.

Buradaki fikir, dizilimleri *k* elemanlı bloklara bölmektir. Verilen i konumundan başlayan, k elemanlı bir pencerenin, bir bloğun sontakısının ve sonraki bloğun öntakısının çarpımı olduğunu gözlemleyin. Böylece, her bloğun öntakısını ve sontakısını aşağıdaki gibi hesaplarız. *i=0,1,...,n-1* için,

$$f_i = \begin{cases} e & \text{if } i \bmod k = 0, \\ f_{i-1} \otimes a_{i-1} & \text{otherwise;} \end{cases}$$

ve *i = n-1,n-2,...,0* için

$$g_i = \begin{cases} a_i & \text{if } (i+1) \bmod k = 0, \\ a_i \otimes f_{i+1} & \text{otherwise.} \end{cases}$$

Bu iki dizilim *Θ(n)* sürede hesaplanabilir ve ardından, i=0,1,....,n-1 için şu çıktı dizilimini elde ederiz.

$$b_i = g_i \otimes f_{i+k}$$

Bu da *Θ(n)* süre alır.

Tek boyutlu 4-filtrelemeyi bir örnek düşünelim.

$$\begin{aligned}
b_0 &= (a_0 \otimes a_1 \otimes a_2 \otimes a_3) &&= g_0 \otimes f_4 \\
b_1 &= (a_1 \otimes a_2 \otimes a_3) \otimes (a_4) &&= g_1 \otimes f_5 \\
b_2 &= (a_2 \otimes a_3) \otimes (a_4 \otimes a_5) &&= g_2 \otimes f_6 \\
b_3 &= (a_3) \otimes (a_4 \otimes a_5 \otimes a_6) &&= g_3 \otimes f_7 \\
b_4 &= (a_4 \otimes a_5 \otimes a_6 \otimes a_7) &&= g_4 \otimes f_8 \\
b_5 &= (a_5 \otimes a_6 \otimes a_7) \otimes (a_8) &&= g_5 \otimes f_9 \\
b_6 &= (a_6 \otimes a_7) \otimes (a_8 \otimes a_9) &&= g_6 \otimes f_{10} \\
&\vdots
\end{aligned}$$

Problem 4. ***ViTo Tasarımı***

ViTo adlı yeni ve geliştirilmiş bir dijital video kaydedici tasarlıyorsunuz. ViTo yazılımında, bir *i* televizyon gösterisi 3'lü olarak simgelenir. Bu 3'lü, **kanal numarası** c_i, **başlangıç zamanı** s_i ve **bitiş zamanı** e_i'dir. ViTo sahibi, *n* tane izlenecek gösteriyi girdi olarak listeler. *i= 1,2,....,n* olan her gösteri bir **keyif reytingi** r_i'ye sahiptir. Gösteriler üstüste çakışabileceğinden ve ViTo aynı anda sadece bir gösteri kaydedebileceğinden, ViTo, alınan keyif reytingi en fazla olan gösterilerin alt kümesini kaydetmelidir. ViTo kullanıcısı, bir gösterinin sadece bir bölümünü seyretmekten zevk almadığından, ViTo hiçbir zaman bölüm kaydı yapmaz. ViTo'nun gösterimlerin en iyi alt kümesini seçebilmesi için verimli bir algoritma yazın.

Çözüm: ViTo'nun programların herhangi bir altkümesini kaydetmek için yeterli sabit disk alanına sahip olduğunu varsayalım. Dinamik programlama yaklaşımı kullanarak bu problemi çözeceğiz.

Öncelikle, en uygun altyapıyı gösterelim. $show_i$, (c_i,s_i,e_i) üçlüsünü ifade etsin; *shows* da $show_1,\ldots ,show_n$ 'lerin tüm listesini ve $shows(t)$ de $e_j < t$ olduğunda $show_j$ 'lerin alt kümesini ifade etsin. (yani bitiş zamanları *t* zamanından önce olan tüm programlar) $show_{i_1}$, $show_{i_2}$, ..., $show_{i_k}$ gibi bir en iyi çözüm düşünelim. Bu durumda, $show_{i_1}$, $show_{i_2}$, ..., $show_{i_{k-1}}$ $shows(s_{i_k})$ altprobleminin en iyi çözüm olmalıdır, çünkü eğer değilse, bu altprobleme daha iyi bir sonucu kesip yapıştırabiliriz; sonuna bir $show_{i_k}$ ekleyerek en iyi olandan daha iyi bir çözüm elde edebiliriz.

Show'ları bitiş sürelerine göre sıralarsak, $i < j$ için $e_i \leq e_j$ olur ve eğer $e_i = e_j$ ise $s_i < s_j$ 'dir. (Eğer 2 veya daha fazla program aynı anda başlar ve biterse, en fazla keyif katsayısı olan r_i 'yi saklar ve bağları rastgele atarız). Bu doğrusal sürede sayma sıralaması ile yapılabilir: n tane programla en fazla 2n muhtemel başlangıç/ bitiş süresi olur ve bir günde sadece 24 saat olduğundan, olası sürelerin aralığı O(n)'dir. Problemleri yeniden etiketleyin, böylece, $show_1, \ldots, show_n$ sıralı düzende olsun. $shows^i$ programların, *i*'inci gösteriye kadar (*i* dahil) listesi olsun; yani, $shows^i = show_1, show_2, \ldots, show_i$

$shows(t)$, $show_j$'nin bir altkümesidir ve bazı $k(t)$'ler, $e_j < t$ olduğunda, ve bunun $show^{k(t)}$'ye eşit olduğuna dikkat edin. En iyi çözümün, $show^i$ programları için birleştirilmiş keyfi p(**i**) olsun;

$$p(i) = \begin{cases} 0 & \text{if } i < 1; \\ \max\{p(k(s_i)) + r_i, p(i-1)\} & \text{otherwise.} \end{cases}$$

En iyi çözüm ise;

$$record(i) = \begin{cases} \{\} & \text{if } i < 1; \\ record(k(s_i)) \cup \{show_i\} & \text{if } p(k(s_i)) + r_i > p(i-1); \\ record(i-1) & \text{otherwise.} \end{cases}$$

Koşma süresi = programları sıralama süresi + $p(n)$'i bulmak için gereken süre = O(n)

Problem 5. ***Bir Grafiği Geliştirmek***

G = (V, E) yönlendirilmiş grafiğini, dinamik olarak geliştiren bir veri yapısı geliştirmek istiyoruz. Başlangıçta, $V = \{1, 2, \ldots, n\}$ ve $E = \varnothing$'ye sahibiz. Kullanıcı, grafiği aşağıdaki işlemi kullanarak geliştirecek.

- INSERT-EDGE(u, v): u köşesinden v köşesine yönlendirilmiş bir kenar ekler. Öyle ki, $E \leftarrow E \cup \{(u,v)\}$'dir.

Ayrıca, kullanıcı istediği zaman iki köşenin bağlantılı olup olmadığını aşağıdaki işlemle sorgulayabilir.

- CHECK-PATH(u, v): Eğer u köşesinden v köşesine bir yönlendirilmiş yol mevcut ise DOĞRU döndür, aksi takdirde YANLIŞ döndür.

Kullanıcı grafiği tamamen bağlantılı olana kadar geliştirmeye devam eder. Kenarların sayısı tekdüze olarak arttığından ve kullanıcı aynı kenarı hiçbir zaman 2 kere araya yerleştirmediğinden, Toplam INSERT-EDGE işlemlerinin sayısı tam olarak n(n-1)'dir. Kullanıcı grafiği geliştirmeye devam ettiği sürece, *m* kere CHECK-PATH işlemini çağırır ve bu işlemler $n(n-1)$ INSERT-EDGE ile birlikte çalışır. Bu tarz işlemler dizisini verimli bir şekilde destekleyecek bir veri yapısı tasarlayın.

Çözüm: Bu problemi çözmek için nxn boyutunda ***transitive-closure*** yani geçişli-kapalı matris *T*'yi tutarız; *T*, her köşe ikilisinin arasında yönlendirilmiş bir yol olup olmadığını saklar. Her bir CHECK-PATH işlemi *O(1)* süre alan ve *n(n-1)* INSERT-EDGE işlemi en kötü durumda $O(n^3)$ süre alan bir algoritma vereceğiz. Bu sınırları bir araya getirirsek m tane CHECK-PATH ve n(n-1) INSERT-EDGE işlemi dizisi toplamda $O(n^3+m)$ süre alır. Sonra *m*'nin küçük olduğu durumlar için veri yapısını geliştirip, $O(min\{ n^3 + m, n^2 m \})$'yi elde edeceğiz.

Veri yapımız geçişli kapalı matris $T=(t_{uv})$'yi destekler ve;

$$t_{uv} = \begin{cases} 1 & : \text{ if there exists a directed path from } u \text{ to } v \text{ in } G \\ 0 & : \text{ otherwise .} \end{cases}$$

T matrisi bir komşuluk matrisine benzer, sadece $u \to v$ kenarları yerine, $u \rightsquigarrow v$ yollarını saklar. *u*'uncu satırdaki 1'lerin *u*'nun erişebileceği bütün köşelere karşılık geldiğine ve *u*'uncu sütundaki 1'lerin de *u*'ya ulaşabilen köşelere karşılık geldiğine dikkat edin. Başlangıçta, $t_{uu} = 1$ olarak ilklendiririz, çünkü bir köşeden kendisine (kenarsız) yönlendirilmiş bir yol vardır.

T verildiğinde, *CHECK-PATH(u,v)* uygulaması açıktır; t_{uv}'nin değerinin sorgulanmasıdır. Bu sorgulama sabit zamanda yapılabilir, bu nedenle CHECK-PATH sabit sürede koşar. CHECK-PATH için sözde kod aşağıdadır.

```
CHECK-PATH(u, v)
1  if t_uv = 1
2      then return TRUE
3      else return FALSE
```

T matrisinin INSERT-EDGE(u,v) ile genişletilmesi veri yapısının karmaşık bölümüdür. *(u,v)* kenarı eklendiğinde, her *x* köşesini kontrol ederiz. Eğer *x, u*'ya erişebiliyorsa ve v'ye erişemiyorsa, matrisimizi, *u*'yu *v*'nin eriştiği her köşeye erişecek şekilde güncelleriz (daha önce erişebildiği köşelere ek olarak). Başka bir deyişle, R_w , w'nun erişebileceği köşelerin kümesi olsun,(yani *T*'deki *w*'ninci sütunda değerleri 1 olan dizinlerinin kümesi). Sonra, *(u,v)*'yi eklerken, bütün *x*$\in$*V*'ler için döngü yaparız. Bütün x'ler için u $\in$ R_x ve v$\notin$$R_x$ olduğunda, $R_x \leftarrow R_x$ u R_v'ye ayarlarız.
INSERT-EDGE için sözde kod aşağıdadır.

```
INSERT-EDGE(u, v)
1  for x ← 1 to n
2      do if t_xu = 1 and t_xv = 0    ▷ x can reach u but not v
3          then for y ← 1 to n
4              do t_xy ← max{t_xy, t_vy}    ▷ If v ⇝ y, add x ⇝ y to T
```

Doğruluk. Aşağıdaki teorem, algoritmamızın doğruluğunu kanıtlar.

Teorem 1 Eğer G grafiğinde, ***iff*** yani eğer ve sadece eğer, x'ten y'ye yönlendirilmiş bir yol varsa, INSERT-EDGE işlemi $t_{xy} = 1$ değişmezini korur.

İspat : INSERT-EDGE işlemleri üzerinde tümevarım ile ispatımızı yaparız. Bir INSERT-EDGE(u,v) işlemine kadar, geçişli-kapalı matrisimizin doğru olduğunu varsayar, sonra, işlemden sonra da doğru olduğunu kanıtlarız. CHECK-PATH işlemi matrisi değiştirmediğinden bu işlem için bir şey ispatlamamız gerekmez.

Önce, (u,v) kenarı eklenmeden önce, $x \rightsquigarrow y$ olduğunu düşünelim. Sonra, INSERT-EDGE işleminden önce $t_{xy} = 1$ olduğunu düşünelim. t_{xy} sadece 4. satırda güncellenebilir ve değeri olan 1 değişmez. Bu yaklaşım doğrudur, çünkü kenarların eklenmesi var olan bir yolu yok etmez.
(u,v) kenarı eklenmeden önce,
$x \not\rightsquigarrow y$ olduğunu,
ama ekleme işleminden sonra
$x \rightsquigarrow y$ olduğunu varsayalım. Bu x'ten y'ye giden yolun (u,v) kenarını kullandığı anlamına gelir. Bu aynı zamanda, INSERT-EDGE(u, v)'den önce

$x \rightsquigarrow u$ ve $v \rightsquigarrow y$

olduğunu gösterir. Varsayımdan yola çıkarak, $t_{xu} = 1$ve $t_{vy} = 1$ demektir. Ayrıca (u, v)'nin eklenmesinden önce $x \not\rightsquigarrow v$ olmalıdır, aksi halde $x \not\rightsquigarrow y$ varsayımını ihlal etmiş oluruz.
Böylece, 4. satıra geldik ve
$t_{xy} \leftarrow t_{vy} = 1$
oldu.
Değerlendireceğimiz son durumda ekleme işleminden sonra,
$x \not\rightsquigarrow y$
olduğunu düşünelim.
Bu durumda $t_{xy} = 0$ olduğundan emin olmalıyız. Eğer yol yoksa, (u,v) eklemesinden önce t_{xy}=0'dır. Ayrıca (u, v)'yi kullanan bir yol olmadığından, ya $t_{xu} = 0$ veya $t_{vy} = 0$'dır. Eğer $t_{xu} = 0$ ise ikinci satırdaki döngüye girmeyiz ve bu nedenle 4. satırdaki güncelleme yapılmaz. Eğer t_{xu}=1 ise, t_{vy}=0 olur ve bu durumda 4. satırda $t_{xy} \leftarrow 0$ değeri elde edilir.

Çözümleme. Şimdi algoritmamızın koşma süresini değerlendirelim. Her CHECK-PATH işlemi sadece tabloya bakmak olduğu için *O(1)* süre alır. Buna karşılık INSERT-EDGE işleminin çözümlemesi biraz daha karmaşıktır. Bu işlemin en kötü durum maliyetini basitçe üst sınır olarak belirleyebilir ve bunun $O(n^2)$ olduğunu söyleyebiliriz, çünkü 4. satırda içiçe ***for*** döngüleri *n* eleman üzerinde sabit iş yapar. Şimdi daha sıkı bir sınırı, *n(n-1)* boyutunda bir dizi INSERT-EDGE işleminin toplu çözümlemesini yaparak belirleyebiliriz.. INSERT-EDGE işlemi her koştuğunda, dış döngü (1. satır), ikinci satırdaki *n* öge için sabit iş yapar. Yani dış döngünün katkısı $O(n^3)$ olur. İç döngü (satır 3) sadece $t_{xv} = 0$ olduğunda işlem yapar ve işi bittiğinde $t_{xv} = 1$ olur. Yani, bir x köşesi iç döngüde en fazla *n* defa (aslında $t_{xx} = 1$ ile başladığımızdan n-1 defa) koşar. Elimizde n köşemiz olduğundan, iç döngümüz $O(n^3)$ miktar iş için en kötü durumda toplamda en fazla n^2 defa koşar. Dolayısıyla toplam koşma süremiz,

n(n-1) INSERT-EDGE ve m CHECK-PATH için $O(n^3+m)$'dir.

Küçük Geliştirmeler. INSERT-EDGE için *O(1)*'e mal olan ancak, CHECK-PATH işlemini $O(n^2)$'de tamamlayan başka bir veri yapısı vardır. Bu veri yapısını geliştirmek için, komşuluk listesini kullanabiliriz: *n* büyüklüğündeki, köşelere göre anahtar listesi olan *A[1...n]* dizilimini kullanabilir ve ilgili köşeden dışarı giden bütün kenarları (zincirli) liste olarak saklayabiliriz. INSERT-EDGE(*u,v*) işlemini gerçekleştirmek için *O(1)* sürede *v*'yi *A[u]*'nun başına ekleriz. (Kenarların sadece bir defa eklendiğini unutmayın, dolayısıyla *v*'nin daha önce listede olup olmadığı için endişelenmemize gerek yok). CHECK-PATH(*u,v*) işlemini gerçekleştirmek için, bir tür arama, örneğin *u* köşesinden başlayarak enine arama yaparız. Eğer v, bu aramanın herhangi bir noktasında karşımıza çıkarsa DOĞRU döndürürüz, aksi takdirde YANLIŞ döndürürüz. Bu algoritmanın doğruluğu bir şekilde oldukça açık olmalıdır. Enine Arama $O(V+E)=O(n^2)$ süre alır. Yani bütün işlemler için toplam koşma süremiz $O(n^2+n^2m)$ olur.

Bu veri yapısı muhtemelen ilk verdiğimizden daha kötüdür. m >> n olduğunu varsaymak, bütün kenarları en az bir kez sorgulayacağımızı düşünürsek oldukça güvenlidir. $m >> n^2$ olduğunu düşünmek bile mantıklıdır. Eğer bu varsayımları yapmak istemezseniz ve m'yi önceden bilirseniz, uygun veri yapısını seçebilirsiniz.

m'nin önceden bilinmediğini düşünsek bile, iki veri yapısını birleştirerek iki sınırın daha iyisini elde edebiliriz. Bunu yapmak için, *n* tane sorgumuz olana kadar komşuluk listesininin veri yapısını kullanırız. *n*'ninci sorguya geldiğimizde (CHECK-PATH), geçici-kapalı matrisi oluştururuz ve bütün sonraki işlemler için bu matrisi kullanırız. Matrisin kurulması *u*'dan başlayan enine arama yaparak ve erişilebilir her *v* köşesini, $t_{uv} \leftarrow 1$ ile işaretleyerek $O(n^3)$ süre alır. Eğer $m \leq n$ ise, sadece komşuluk listesini kullanarak toplam koşma süresi $O(n^2m)$'yi elde ederiz.
Eğer $m \geq n$ ise, önce $O(n^3)$ iş için komşuluk listesini kullanırız ve geçişli-kapalı matrisi $O(n^3)$ sürede dönüştürerek, sonraki tüm işlemler için bu matrisi kullanır ve $O(n^3+m)$ elde ederiz.. Böylece bu veri yapısının koşma süresi en kötü durumda, toplamda $O(\min\{n^3+m, n^2m\})$ olur.