

# C++ Standart Şablon Kütüphanesi'ndeki Dinamik Veri Yapılarının Seyrek Matris İşlemlerindeki Performansları Üzerine Bir İnceleme

Barış Bekmez, A. Egemen Yılmaz  
Ankara Üniversitesi  
Elektrik-Elektronik Mühendisliği Bölümü  
Ankara  
[barisbekmez@gmail.com](mailto:barisbekmez@gmail.com), [aeyilmaz@eng.ankara.edu.tr](mailto:aeyilmaz@eng.ankara.edu.tr)

Mustafa Kuzuoğlu  
Orta Doğu Teknik Üniversitesi  
Elektrik-Elektronik Mühendisliği Bölümü  
Ankara  
[kuzuoglu@metu.edu.tr](mailto:kuzuoglu@metu.edu.tr)

**Özet:** Sonlu Elemanlar Yöntemi kullanımında ihtiyaç duyulan seyrek matrislerin bilgisayarda bellek kazancı sebebiyle sıkıştırılmış tutulması gerekmektedir. Bu sıkışmış matrislerin bellekte saklanması dinamik veri yapıları kullanımı özellikle önem arz etmektedir. Sistemin çözümünde kullanılacak bu seyrek matrisin  $Ax=b$  tipi bir denklemde yer alması, ve bu denklemi oluşturan bazı köklerin işlem öncesi biliniyor olması işlem sayısını azaltmak adına bu matris üzerinden ilgili satır ve sütunların silinebilmesini mümkün kılmaktadır. Çalışmada seyrek matris üzerinden satır-sütun silme işlemlerinde dinamik veri yapılarından “vector”, “list” ve “deque” incelenmiş, bu yapıların çeşitli parametrelerle oluşturulan ve dört değişik sıkıştırma yöntemi ile farklı formatlarda sıkıştırılmış matrisler üzerindeki performansları gözlenmiştir.

## 1. Giriş

Herhangi bir problemin Sonlu Elemanlar Yöntemi ile çözümü sürecinde elde edilen ve sistemi betimleyen  $Ax=b$  denklemi  $A$  matrisi, seyrek bir matristir. Bu nedenle, bilgisayar kaynaklarının etkin kullanılması adına,  $A$  matrisinin hafızada özel bir takım formatlarda saklanması gerekmektedir.

Öte yandan, problem uzayı içerisindeki fiziksel koşul ve kuralların tanımlanması sonucu türetilen sınır koşullarının uygulanması esnasında da; söz konusu  $x$  vektöründeki bilinen bir takım değerler yerine konulacağı için,  $Ax=b$  denklemi  $A$  matris ve vektörlerin satır/sütunlarının silinerek boyutlarının indirgenmesi gerekmektedir. Dolayısıyla, söz konusu matris ve vektörlerin saklanması esnasında, çalışma zamanı içerisinde boyut değiştirmeye olanak sağlayan dinamik veri yapılarının kullanılması da büyük önem arz etmektedir. Örneğin,  $Ax = b$  matris denkleminin, Denklem (1)'deki gibi olması:

$$\begin{bmatrix} a_{11} & 0 & a_{13} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & 0 & 0 & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \quad (1)$$

ve bu matris denkleminde bilinmeyenlerden herhangi birinin (örneğin  $x_3$ 'ün) sınır koşulları vb. herhangi bir şekilde değerinin biliniyor olması durumunda söz konusu matris denklemini:

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & a_{32} & a_{34} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 - a_{13}x_3 \\ b_2 \\ b_3 - a_{33}x_3 \\ b_4 \\ b_5 \end{bmatrix} \quad (2)$$

şeklinde yazılabilir. Dikkat edildiğinde,  $x_3$ 'ün değerinin biliniyor olması durumunda, **A** matrisinin 3. sütununun silindiği; bu sütunda bulunan 0'dan farklı değerlerin  $x_3$  ile çarpılarak eşitliğin sağ tarafına atıldığı görülmektedir. Bu durumda, eldeki denklem sayısı, artık bilinmeyen sayısından fazla olduğu için denklemlerden herhangi bir tanesi kullanılmadan da çözüm elde edilebilmektedir. Sistematik olmak adına, örneğin 3 numaralı denklemin kullanılmamasına karar verilirse, bu durum **A** matrisi ile **x** ve **b** vektörlerinin 3. satırlarının da silinmesi anlamına gelecektir.

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & 0 & a_{45} \\ 0 & 0 & a_{54} & a_{55} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 - a_{13}x_3 \\ x_2 \\ x_4 \\ x_5 \end{bmatrix} \quad (3)$$

Birçok programlama dili, dinamik veri yapılarına olanak sağlamamaktadır; bu nedenle, geliştirilen birçok uygulamada hafızada tutulması gereken veri boyutlarının üst sınırları hesaplanmakta ve matris/vektörlerin saklanması amacıyla hafızada gereğinden büyük alanlar ayrılmaktadır. Ancak, C++ programlama dili ve bu dil ile birlikte tanımlanmış/geliştirilmiş olan Standart Şablon Kütüphanesi (*Standard Template Library* – STL), dinamik veri yapılarını desteklemektedir. Matris ve vektör tarzı yapıları hafızada dinamik olarak saklamak adına STL tarafından sağlanan başlıca veri yapıları *vector*, *deque* ve *list*'tir. Bu yapıların her birinin, elemanlarına sıralı olarak veya rastgele ulaşma, baş / orta / son kısmına yeni veri ekleme (veya buralardan veri çıkarma) vb. hususlarda bir takım avantaj ve dezavantajları bulunmaktadır.

## 2. Seyrek Matris Depolama Yöntemleri

Bir çok elemanında 0 değeri olan matrisler, seyrek matris olarak anılmaktadır. Bilimsel hesaplamalarda,  $N \times N$  boyutlarındaki bir seyrek matrisin, sadece 0'dan farklı elemanlarının konumlarını ve değerlerini tutmak, depolamada  $O(N)$ , çözüm esnasında da  $O(N^2)$ 'ye varan verimlilikte yöntemleri mümkün kılmaktadır. Literatürdeki seyrek matris depolama yöntemlerinden Koordinat (*Coordinate* - COO) yöntemi [1], ELLPACK yöntemi [2], Bentley yöntemi [3], Multifrontal yöntemi [4], bu çalışmanın kapsamına dahil edilmiştir.

Denklem (1)'de görülmekte olan **A** matrisi, Koordinat (*Coordinate* – COO) yönteminde [1] aşağıda belirtilen diziler aracılığıyla hafızada tutulmaktadır

$$\begin{aligned} ri &= [1 \ 1 \ 2 \ 3 \ 3 \ 3 \ 4 \ 5 \ 5] \\ ci &= [1 \ 3 \ 2 \ 2 \ 3 \ 4 \ 5 \ 4 \ 5] \\ val &= [a_{11} \ a_{13} \ a_{22} \ a_{32} \ a_{33} \ a_{34} \ a_{45} \ a_{54} \ a_{55}] \end{aligned}$$

Aynı matris, ELLPACK olarak bilinen seyrek matris depolama yönteminde [2], şu dizilerde tutulmaktadır:

$$val = \begin{bmatrix} a_{11} & a_{13} & * \\ a_{22} & * & * \\ a_{32} & a_{33} & a_{34} \\ a_{45} & * & * \\ a_{54} & * & * \end{bmatrix}, \quad ind = \begin{bmatrix} 1 & 3 & * \\ 2 & * & * \\ 2 & 3 & 4 \\ 5 & * & * \\ 4 & * & * \end{bmatrix}$$

Bentley seyrek matris depolama yönteminde [3] ise aynı matris, aşağıda belirtilen dizilerde tutulmaktadır:

$$ija = [7 \ 8 \ 8 \ 10 \ 11 \ 12 \ 3 \ 2 \ 4 \ 5 \ 4]$$

$$sa = [a_{11} \ a_{22} \ a_{33} \ 0 \ a_{55} \ * \ a_{13} \ a_{32} \ a_{34} \ a_{45} \ a_{54}]$$

Multifrontal yönteminde [4] ise söz konusu diziler şu şekildedir:

$$ia = [1 \ 3 \ 4 \ 7 \ 8 \ 10]$$

$$ja = [1 \ 3 \ 2 \ 2 \ 3 \ 4 \ 5 \ 4 \ 5]$$

$$a = [a_{11} \ a_{13} \ a_{22} \ a_{32} \ a_{33} \ a_{34} \ a_{45} \ a_{54} \ a_{55}]$$

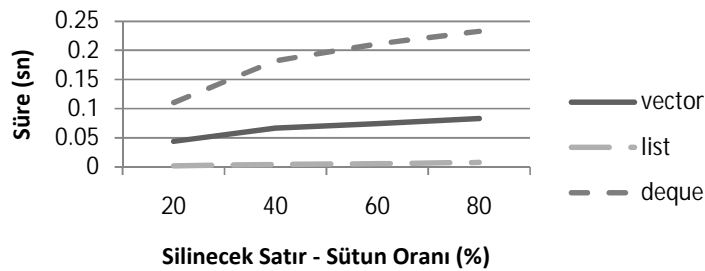
Yukarıdaki dizilerde “\*” ile gösterilen terimler, kasıtlı olarak boş bırakılmış ve değeri önemsiz olan terimlerdir.

### 3. Yapılan Analizler ve Bulgular

Yapılan analizlerde, önceki bölümde anılan 4 farklı seyrek matris depolama yöntemi için, 3 temel STL veri yapısının (*vector*, *list*, *deque*) kullanılması durumunda matris üzerindeki satır/sütun silme işlemlerinin işlemsel maliyetleri hesaplanmıştır. Bu amaçla her bir seyrek matris depolama yöntemi, C++ Standart Şablon Kütüphanesi’ndeki 3 adet dinamik veri yapısı (*vector*, *list*, *deque*) ile ayrı ayrı gerçekleştirilmiş; her bir implementasyon için, satır silme işleminin zamansal maliyeti (matris boyutu, matris doluluk oranı ve silinen satır oranı gibi parametrelere de bağlı olacak şekilde) karşılaştırılarak değerlendirilmiştir. Performans (zaman) ölçümleri, belirli boyut ve doluluk oranına sahip olacak şekilde rastgele oluşturulmuş seyrek matrislerde, belirli bir oranda ancak rastgele belirlenmiş satırlar silinerek gerçekleştirilmiş; sonuçlar, bağımsız 100 adet Monte Carlo koşusu sonucu elde edilen değerlerin ortalaması üzerinden değerlendirilmiştir.

Çalışma kapsamında seyrek matrislerle uğraşılıyor olunması nedeniyle oluşturulacak bu seyrek matrisler %50 doluluk oranına kadar %10’luk dilimlerde incelenmiştir. Kullanılan C++ derleyicilerinde bulunan belirli bellek sınırlamaları nedeniyle üzerinde çalışma yapılacak seyrek matrislerde maksimum 1000×1000’lik boyuta kadar çıkılabilmiş, bu sınırlama sebebiyle çalışmada 250×250, 500×500, 750×750 ve 1000×1000 boyutlarında seyrek matrisler oluşturularak bu boyutlardaki matrislerin üzerinde %20 artışı dilimlerde silme süreleri hesaplanmıştır.

Koordinat (*Coordinate* - COO) yöntemi için elde edilen süreler, bu yöntemde seyrek matris ile ilgili her parametrede *list* veri yapısının süre avantajı sağladığını göstermektedir. Şekil 1’de COO sıkıştırma yönteminde 250×250 boyutlu %10 dolulukta bir seyrek matrisin veri yapısı - süre ilişkileri örnek olarak gösterilmektedir.



Şekil 1. COO yöntemi 250×250 boyutlu matriste %10 doluluk oranında satır-sütun silme süreleri

Öte yandan, ELLPACK, Bentley ve Multifrontal sıkıştırma yöntemleri için alınan sonuçlarda, yöntem bazında birbirlerinden farklı süreler elde edilmiş olsa da, üç sıkıştırma yönteminde sonuçların çok tutarlı olduğu gözlenmiştir. Üç yöntemde de tüm parametrelerde *vector* veri yapısı bariz şekilde süre avantajı sağlamaktadır. Matris boyutunun artması sonucu bu üç sıkıştırma yönteminde de *list* veri yapısının diğerlerinden oldukça fazla süreler vermesi sebebiyle bu süreler grafiklendirilememiş, ancak süreler çizelgelerle incelenebilmiştir.

Yöntemlerin süre ölçümleri Çizelge 1, Çizelge 2 ve Çizelge 3’te sunulmaktadır.

**Çizelge 1.** ELLPACK yönteminde 500×500 boyutlu matriste bazı satır-sütun silme süreleri

Matris Boyutu ( $N \times N$ )	Her Satırda Maks. Eleman Sayısı	Silinecek Satır Oranı (%)	Silme Süresi (sn)		
			<i>vector</i>	<i>list</i>	<i>deque</i>
500	100	20	1.169189	1741.714913	6.476372
500	100	40	2.052651	3149.538740	11.190543
500	100	60	2.776870	4468.707431	13.474415
500	100	80	2.984718	10000.00000+	15.839647

**Çizelge 2.** Bentley yönteminde 250×250 boyutlu matriste bazı satır-sütun silme süreleri

Matris Boyutu ( $N \times N$ )	Her Satırda Maks. Eleman Sayısı	Silinecek Satır Oranı (%)	Silme Süresi (sn)		
			<i>vector</i>	<i>list</i>	<i>deque</i>
250	20	20	0.145034	20.910252	0.703716
250	20	40	0.221134	28.409906	0.971694
250	20	60	0.258942	32.122212	1.101881
250	20	80	0.264767	32.618141	1.015701

**Çizelge 3.** Multifrontal yönteminde 1000×1000 boyutlu matriste bazı satır-sütun silme süreleri

Matris Boyutu ( $N \times N$ )	Her Satırda Maks. Eleman Sayısı	Silinecek Satır Oranı (%)	Silme Süresi (sn)		
			<i>vector</i>	<i>list</i>	<i>deque</i>
1000	40	20	31.147106	126.235347	434.827166
1000	40	40	53.442597	147.834714	770.252454
1000	40	60	68.549169	136.929628	1006.119360
1000	40	80	77.565217	147.901032	1156.626691

#### 4. Sonuç ve Tartışma

Bu bildiride verilen örnekler rasgele seçilmiş olup, çalışmada görülen sabit sonuçları temsil etmektedirler. Çalışmada alınan sonuçlara göre; incelenmiş 4 sıkıştırma yönteminin bilgisayar üzerinde dinamik veri yapıları aracılığıyla kullanılması durumunda Koordinat yönteminde *list* veri yapısının, ELLPACK, Bentley ve Multifrontal yöntemlerinde ise *vector* veri yapısının süre avantajı sağladığı anlaşılmıştır.

İleride yapılacak çalışmalarda, Sıkıştırılmış Seyrek Satır (*Compressed Sparse Row – CSR*), Skyline vb. farklı yöntemlerin de kapsama alınması; matris denkleminin iteratif veya direkt yöntemlerle çözümü esnasında da hangi sıkıştırma yönteminin daha avantajlı olduğunun incelenmesi hedeflenmektedir. Bu tür çalışmaların özellikle nümerik elektromanyetik, daha spesifik olarak da Sonlu Elemanlar Yöntemi konusunda çalışmalar yürüten araştırmacılara hangi tür matris depolama yönteminde hangi tür dinamik veri yapılarının kullanılması gerektiği konusunda ışık tutacağı değerlendirilmektedir.

#### Kaynaklar

- [1] Anonim, “Numpy and Scipy Documentation”, Çevrimiçi: <http://docs.scipy.org/doc/>, 2008.
- [2] Rice, J. R. ve Boisvert, R. F., Solving Elliptic Problems Using ELLPACK, Springer Verlag, 1984.
- [3] Bentley, J., Programming Pearls, MA: Addison-Wesley, 1986.
- [4] Irons, B. M., “A frontal solution program for finite-element analysis”, International Journal of Numerical Methods in Engineering, cilt 2, s. 5-32, 1970.