ივანე ჯავახიშვილის სახელობის თბილისის
სახელმწიფო უნივერსიტეტი

ვახტანგი ლალუაშვილი

მეჩხერი მატრიცები, მათი იმპლემენტაციები და გამოყენებები

ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერება

დოქტორანტის სემინარი I

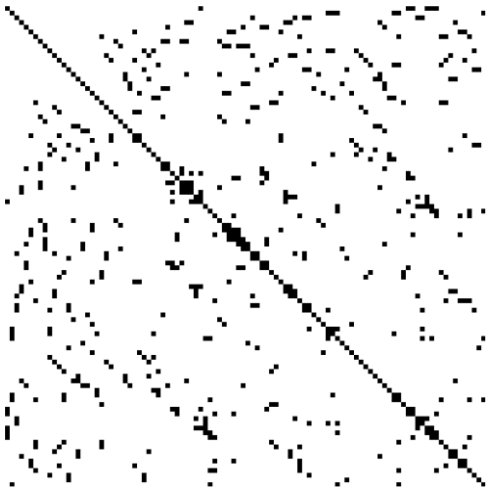
ხელმძღვანელი: პროფესორი კობა გელაშვილი

თბილისი, 2018

შესავალი

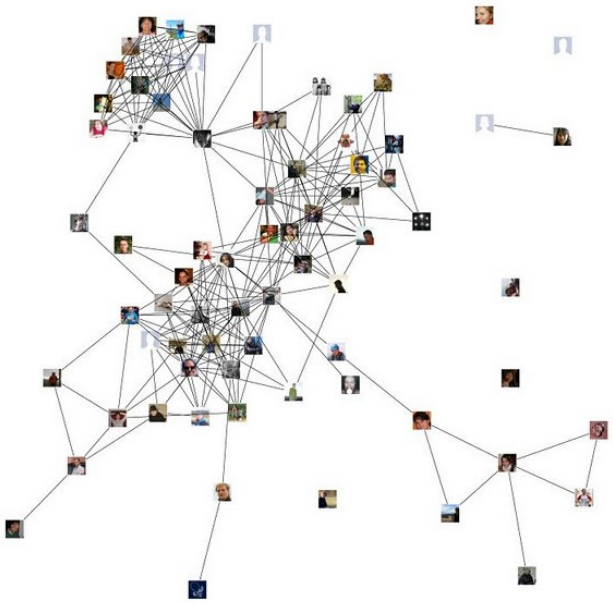
ზოგადად, სიმეჩხერე გულისხმობს მონაცემებში არანულოვან ელემენტებთან შედარებით ნულების სიჭარბეს, მაგრამ ეს სიმეჩხერის საზღვარი გაურკვეველია და დამოკიდებულია სხვადასხვა ფაქტორზე. მაგალითად, მეჩხერი მატრიცი განიმარტება როგორც მატრიცი, რომელსაც აქვს ცოტა არანულოვანი ელემენტი და ეწოდება მკვრივი წინააღმდეგ შემთხვევაში. ზოგიერთი განმარტების მიხედვით, კვადრატულ მატრიცას n სტრიქონით ეწოდება მეჩხერი, თუ მასში არანულოვანი ელემენტების რაოდენობა არის O(n) რიგის. [5]-ში გამოთქმულია მოსაზრება, რომ მატრიცს შეიძლება ეწოდოს მეჩხერი იმ შემთხვევაშიც, როდესაც შესაძლებელია რაიმე სპეციალური მეთოდების გამოყენებით დიდი რაოდენობის ნულებისგან სარგებელის მიღება (მაგალითად, რაიმე ამოცანაში ოპერაციების დაჩქარება). ძირითადი იდეა მეჩხერი მატრიცის ასეთ მეთოდებში მდგომარეობს იმაში, რომ არ დაგვჭირდეს ნულოვანი ელემენტების შენახვა. მთავარი ამოცანაა განისაზღვროს მონაცემთა სტრუქტურები ასეთი მატრიცებისთვის, რომლებიც კარგად და ეფექტურადაა მორგებული სტანდარტულ ამოცანების ამომხსნელ მეთოდებზე: პირდაპირ ან იტერაციულზე.

ამ ყველაფრიდან ჩანს, რომ მეჩხერი მატრიცის „აბსოლუტური“ განმარტება არ არსებობს. მატრიცის სიმეჩხერე ფარდობითია და დამოკიდებულია როგორც ალგორითმზე, რომელიც იყენებს მეჩხერ მატრიცს, ასევე მატრიცის სტრუქტურაზე და იმპლემენტაციაზე. მაგალითად შეიძლება მოვიყვანოთ ჩვენს სტატიაში ([6]-ში) განხილული მეჩხერი მატრიცის (JNZ) ფორმატისთვის ჩატარებული ექსპერიმენტი, სადაც განხილულ იქნა 15 სხვადასხვა ზომის მართკუთხა მატრიცა სტრიქონების რაოდენობით n (n იცვლება 100-იდან 1500-ის ჩათვვით, ბიჯით 100). თითოეული n-ისთვის, შედგენილ იქნა შემთხვევითი რიცხვებით შევსებული 5 განსხვავებული ვარიანტი, ნულების პროცენტული შემადგენლობით 15, 20, 25, 30 და 35. სულ 90 განსხვავებული მატრიცა. ყოველი მატრიცისთვის შემთხვევითი რიცხვებით დაგენერირდა შესაბამისი განზომილების ვექტორი. მატრიცი-ვექტორის ყოველი წყვილისთვის, CG (Conjugate Gradient) მეთოდით ამოხსნილ იქნა სისტემა. ტესტების შედეგებმა აჩვენა, რომ 35%-იანი ნულების შემთხვევაში, აბსოლუტურ უმრავლესობაში JNZ ფორმატის გამოყენება აჩქარებდა სისტემის ამოხსნას. ნულების რაოდენობის ზრდის კვალობაზე, განსხვავება კიდევ უფრო თვალსაჩინო ხდება.



სურათი 1. შავი პიქსელები წარმოადგენს არანულოვანი ელემენტებს.

საზოგადოდ, მეჩხერი მატრიცები გვხვდება სხვადასხვა გამოყენებებში: გრაფებთან დაკავშირებულ ალგორითმებში, კერძოწარმოებულიანი განტოლებების ამოხსნის პროცესში, სადაც მატრიცები ყოველთვის შეიცავს გარკვეული რაოდენობის ნულოვან ელემენტებს და ა. შ. მაგალითად, კომპანია Google-ი თავის საძიებო სისტემაში მეჩხერ მატრიცებს იყენებს ვებ-გვერდების ერთმანეთთან დაკავშირებების შესანახად (იგულისხმება სხვადასხვა მისამართის მქონე ვებ-გვერდები). ამ შემთხვევაში საქმე გვაქვს მეჩხერ გრაფთან, სადაც წიბოების რაოდენობა საგრძნობლად ცოტაა. მატრიცში i-ური ვებ-გვერდის კავშირი j-ურთან შეიძლება 1-იანით გამოვსახოთ. აგრეთვე, კომპანია Facebook-იც, მსგავსად Google-ისა, იყენებს მეჩხერ მატრიცებს „მეგობრობის“ ინფორმაციის შესანახად: ვინ-ვისთან მეგობრობს.



სურათი 2. მეგობრობის გრაფის ფრაგმენტი.

მეჩხერი მატრიცის ფორმატები

პროგრამის წერისას უბრალო მატრიცის მეხსიერებაში შენახვა, ყველაზე პრიმიტიულად, შეგვიძლია ორგანზომილებიან მასივში (ან ერთგანზომილებიან ვექტორში სტრიქონების მიმდევრობით ჩაყრით):

```
double m[][2] = { {1, 2}, {3, 4} };

// ანდა დინამიკურად
double* matrix = (double*) malloc(nrows * ncols * sizeof(double));
```

მსგავსი მიდგომა მჭიდრო (dense) მატრიცის შემთხვევაში პრობლემას არ წარმოადგენს, მაგრამ მეჩხერი მატრიცის დროს ვაწყდებით ორ პრობლემას: ზედმეტად ვიყენებთ მეხსიერებას 0-ების შესანახად, ხოლო მატრიცულ ოპერაციებში ასეთი 0-ები მაინც იღებს მონაწილეობას, ტუმცა შედეგს არ ცვლის. ოპტიმიზებული კომპილერები შესაზლოა ხვდებიან რომ ნულზე ოპერაცია არ განახორციელონ, მაგრამ მატრიცის ყოველ პოზიციაზე მიმართვა მაინც ცვლის ასიმპტოტიკას.

2x2 მატრიცის შემთხვევაში მეხსიერების პრობლემა ნაკლებად შესამჩნევია, მაგრამ თუ მასივს ავიღებთ 1,000,000 x 1,000,000-ს, მაშინ ვნახავთ, რომ ეკონომიურად არ ვიყენებთ მეხსიერებას და ვანელებთ პროგრამას.

მაგალითად, თუ ავიღებთ `double` ტიპის მეჩხერ მატრიცს 1მილ. x 1მილ. რაოდენობის მონაცემებით, მაშინ მოგვიწევს ოპერატიული მეხსიერების 7.27596 ტერაბაიტი მოცულობის გამოყენება. რა თქმა უნდა თანამედროვე სამყაროში უკვე არსებობს 1TB-იანი მეხსიერების ბარათები (და ახლო მომავალში უფრო დიდებიც იარსებებს), მაგრამ უმრავლესობას დღესდღეობით მხოლოდ რამდენიმე გიგაბაიტი აქვს. აგრეთვე არ დაგვავიწყდეს ის ფაქტი, რომ მეხსიერების გაზრდასთან ერთად გაგვეზრდება მატრიცის გამოყენებით ალგებრული ოპერაციების შესრულების დროც. მაგალითად, მატრიცის ვექტორზე გამრავლების დროს: დრო დაგვეკარგება ვექტორის ელემენტის ნულზე გამრავლებისას.

ამ პრობლემების მოსაგვარებლად გვაქვს ე. წ. მეჩხერი მატრიცების ფორმატები. ფორმატები იმიტომ, რომ არ არსებობს მხოლოდ ერთი უნიკალური მეჩხერი მატრიცის ფორმატი, რომელიც ოპტიმალური იქნება ყველა სახის მეჩხერი მატრიცისთვის.

საერთოდ, მეჩხერი მატრიცების ინტენსიური შესწავლა მიმდინარეობს 1970-იანი წლებიდან. ამ პერიოდში შეიქმნა და დამუშავდა რამდენიმე მონაცემთა სტრუქტურა (ფორმატი) მათი წარმოდგენისთვის (და ახლაც იქმნება და მუშავდება). ზოგიერთი ფორმატი დამუშავებულია ისეთი შემთხვევებისთვის, როდესაც სიმეჩხერე ვლინდება გარკვეული სისტემატიკური მოდელის სახით (მაგალითად, სურათი 1. დიაგონალური), ან როდესაც არანულოვანი ელემენტების განლაგება არ ექვემდებარება რაიმე კანონზომიერებას. ვხვდებით როგორც მარტივ - ასევე კომპლექსურ ფორმატებს, როგორც აღქმის მხრივ - აგრეთვე წარმოდგენისაც.

მეჩხერი მატრიცის ფორმატები ორ კატეგორიად შეიძლება გაიყოს: Point Entry და Block Entry ფორმატად. Point Entry ფორმატი გულისხმობს, რომ ყოველი ელემენტი მატრიცის ელემენტს წარმოადგენს. ხოლო, Block Entry გულისხმობს, რომ ყოველი ელემენტი განსაზღვრავს ელემენტების ნებისმიერი ზომის მკვრივ ორგანზომილებიან ბლოკს.

მეჩხერი მატრიცის ფორმატებია:

- **DNS** – Denses
- **BND** – Linpack Banded
- **COO** – Coordinate
- **CSR** – Compressed Sparse Row
- **CSC** – Compressed Sparse Column
- **MSR** – Modified CSR
- **LIL** – Lined List (list of lists)
- **ELL** – Ellpack-Itpack
- **DIA** – Diagonal
- **BSR** – Block Sparse Row
- **SSK** – Symmetric Skyline
- **NSK** – Nonsymmetric Skyline
- **JAD** – Jagged Diagonal (or JDS)
- **SELL** – Sliced ELL
- **BSRX** – Extended BSR
- **HYB** – Hybrid
- **TJDS** – Transposed Jagged Diagonal
- **Bi-JDS** – Bi Jagged Diagonal
- **DOK** – Dictionar of Keys
- **JSA** – Java Sparse Array
- **JNZ** – Jagged Non-zero

მაგალითად ამ სიიდან, Point Entry კატეგორიას შეიძლება მივაკუთვნოთ COO, CSR ფორმატები ხოლო Block Entry კატეგორიას BSR ფორმატი და ა. შ.

ცხადია, ეს სია არასრულია და ზოგიერთი აღწერილი ფორმატი სხვა არსებული ფორმატის მოდიფიცირებული ვარიანტია. მაგალითად, TJDS და Bi-JDS არის JDS-ის ალტერნატიული ვარიანტები. აგრეთვე, MSR წარმოადგენს მოდიფიცირებულ CSR და CSC იგივე CSR არის ერთი პატარა ცვლილებით. საზოგადოდ, ყველაზე ცნობილი ფორმატია CSR და უმეტესობა სხვა ფორმატებისა იყენებს CSR-ის პრინციპს. JSA და JNZ წარმოადგენენ ELL ფორმატის განზოგადებებს.

განვიხილოთ რამდენიმე მნიშვნელოვანი ფორმატი: COO, CSR, BSR, DIA, ELL, JSA და JNZ.

COO (Coordinate) ფორმატი

ყველაზე მარტივი ფორმატი (რომელიც მეხსიერების ეკონომიის შესაძლებლობას გვთავაზობს) არის კოორდინატული ფორმატი, Coordinate format (COO). COO-ში მატრიცი გამოისახება სამი მასივის სახით: სტრიქონების მასივი, სვეტების მასივი და მნიშვნელობების მასივი. ყოველი არანულოვანი ელემენტისთვის გვაქვს სამი ჩანაწერი i-ურ პოზიციაზე: სტრიქონის ინდექტი, სვეტის ინდექსი და ელემენტის მნიშვნელობა. ვთქვათ, გვაქვს შემდეგი მატრიცი:

[**1.0** **2.0**]

[0.0 **4.0**]

ამ მატრიცის შენახვა COO ფორმატში გვაძლევს შემდეგ სამ მასივს:

```
unsigned rows[3] = { 0, 0, 1 };
```

```
unsigned columns[3] = { 0, 1, 1 };
double values[3] = { 1.0, 2.0, 4.0 };
```

სადაც ვხედავთ, რომ (0, 0) პოზიციაზე 1.0 მნიშვნელობა წერია, (0, 1)-ზე 2.0 და ა. შ. შევნიშნოთ, რომ ნულის მნიშვნელობას (და მის პოზიციას) არ ვინახავთ.

ჩანაწერის მოძებნა ხდება მარტივი იტერაციით. მაგალითად, იმისათვის, რომ (1, 1) პოზიციაზე მყოფ ელემენტის მნიშვნელობა ვიპოვოთ უნდა ყოველი ელემენტისთვის შევამოწმოთ შემდეგი ორი პირობა: `if (rows[i] == 1 && columns[i] == 1)`. თუ პირობა შესრულდა, მაშინ მნიშვნელობა ჩაწერილია მნიშვნელობების i-ურ ინდექსზე, თუ არადა - მნიშვნელობა ყოფილა 0.0.

დავუბრუნდეთ ჩვენს 1,000,000 x 1,000,000 ზომის მეჩხერ მატრიცს და დავუშვათ, რომ 5% ელემენტებისა არანულოვანია (არანულოვანები აღინიშნება nnz-თი). აქედან გამოდის, რომ nnz=50,000,000,000. COO ფორმატის გამოყენებისას (და თუ ჩავთვლით, რომ `sizeof(unsigned) == 4`) ჩვენი გამოყენებული მეხსიერების ზომა იქნება:

```
(sizeof(unsigned) + sizeof(unsigned) + sizeof(double)) * 5^10
```

რაც არის 800 GB და საგრძნობლად უფრო ნაკლებია ვიდრე თავდაპირველი 8 TB, რაც მკვრივი ფორმატის დროს მივიღეთ.

ჩვენ შეგვიძლია ვიპოვოთ საზღვარი, როდესაც COO ფორმატის გამოყენება მკვრივ ფორმატთან შედარებით ამცირებს გამოყენებულ მეხსიერებას. ამისათვის გავარკვიოთ როდის არის არანულოვანებისთვის 16 ბაიტის შენახვა ნაკლები ყველა ელემენტისთვის 8 ბაიტის შენახვაზე:

```
16 * nnz < 8 * nrows * ncols
```

ასევე, თუ განვსაზღვრავთ სიმეჩხერეს, როგორც პროცენტს არანულოვანი ელემენტებისა, მაშინ მივიღებთ:

```
16 * (sparsity * nrows * ncols)   <   8 * nrows * ncols
      sparsity * nrows * ncols    <   0.5 * nrows * ncols
                      sparsity    <  0.5
```

აქედან გამომდინარე, როდესაც მატრიცის სიმეჩხერე 50%-ზე ნაკლებია (მნიშვნელობების ნახევარზე ნაკლები არანულოვანებია), მაშინ ჩვენ შევძლებთ მეხსიერების დაზოგვას მონაცემების COO ფორმატში შენახვით. შევნიშნოთ, რომ ეს ფორმულა მივიღეთ `double` ტიპის მონაცემისთვის. უფრო ზოგადი სახის ფორმულას აქვს შემდეგი სახე:

$$\text{sparsity} < \frac{\text{\# of bytes per value in dense}}{\text{\# of bytes per value in COO}}$$

მიუხედავად იმისა, რომ მეხსიერება შეგვიმცირდა, COO ფორმატმა წარმოქმნა რამდენიმე პრობლემა. პირველი არის ის, რომ თითოეული არანულოვანი მონაცემის შესანახი მეხსიერება გაგვიორმაგდა (ადრე იყო 8 ბაიტი ახლა არის 16 ბაიტი). ჯამში მეხსიერება შევამცირეთ, მაგრამ თითო ელემენტზე გამოყოფილი 16 ბაიტი არ ჰგავს იდეალურ

გადაწყვეტილებას. აგრეთვე, მნიშვნელობის ამოღების ოპერაცია COO ფორმატში უარეს შემთხვევაში მოითხოვს ყველა არანულოვანი ელემენტის გავლას (თუ ინდექსები დალაგებული არ გვაქვს). ხოლო მკვრივ ფორმატში, მაგალითად, (980, 1020) ინდექსზე მყოფი ელემენტის ამოღება ხდება პირდაპირ, 1 ბიჯში.

CSR (Compressed Sparse Row) ფორმატი

CSR ფორმატი წარმოადგენს COO-ს მარტივ გაუმჯობესებას. COO-ს ერთ-ერთი ნაკლი იყო მისი სტრიქონებისა და სვეტების ინდექსების შენახვის სტრატეგია. ვთქვათ, გაქვთ 3x3 მეჩხერი მატრიცი, რომელსაც ვინახავთ COO ფორმატში:

[**3.0** 0.0 0.0 **4.0** 0.0 **1.0**]

[**1.0** 0.0 0.0 0.0 0.0 0.0]

[0.0 0.0 0.0 **2.0** 0.0 0.0]

...

```
unsigned rows[5] = { 0,   0,   0,   1,   2 };
unsigned columns[5] = { 0,   3,   5,   0,   3 };
double   values[5] = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

ჩანს, რომ ვინახავთ ზედმეტ მონაცემებს სტრიქონების მასივში: 5 მნიშვნელობას ვინახავთ სტრიქონის სამი მნიშვნელობის წარმოსაჩენად (სტრიქონი: 0, 1, 2). ჩვენ შეგვიძლია სტრიქონების შემჭიდროვება, თუ დავალაგებთ სვეტებს და მნიშვნელობებს სტრიქონის მიხედვით, და სტრიქონების მასივში სტრიქონში არსებული არანულოვანი ელემენტების რაოდენობას შევინახავთ (და არა ელემენტების ინდექებს):

```
unsigned row_counts[3] = { 3,             1,   1 };
unsigned columns[5] = { 0,   3,   5,   0,   3 };
double   values[5] = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

ამით დავზოგეთ ორი რიცხვი. ამჯერად, row_counts[i] გვეუბნება რამდენი არანულოვანი ელემენტია i-ურ სტრიქონში, ამიტომაც row_counts უნდა იყოს სტრიქონების რაოდენობის ტოლი და არა არანულოვანი ელემენტების რაოდენობისა. მანამ, სანამ თითოეულ სტრიქონში არანულოვანი ელემენტების რაოდენობა საშუალოდ იქნება > 1, ჩვენ დავზოგავთ მეხსიერებას.

ვთქვათ, გვსურს ასეთ ფორმატში (2, 3) ელემენტის მნიშვნელობის გაგება. თავდაპირველად უნდა ვიპოვოთ სტრიქონი 2 სვეტებისა და მნიშვნელობების მასივში საიდან იწყება. ამის გაგება შეგვიძლია მარტივად, რადგანაც ვიცით, რომ მეორე სვეტის მნიშვნელობები იწყება იქიდან სადაც მთავრდება წინა სტრიქონის ელემენტები:

sum(row_counts[0 : i - 1])

ამიტომაც, მე-2 სტრიქონის დასაწყისის საპოვნელად უნდა ავჯამოთ row_counts[0] და row_counts[1] და მივიღებთ 4-ს. შემდეგ სვეტების მასივში ვნახავთ სვეტის ინდექსს და შესაბამის პოზიციაზე მოვძებნით მნიშვნელობას.

იმ შემთხვევაში, როდესაც მასივი დიდია სტრიქონების არანულოვანი ელემენტების აჯამვა დიდ გამოთვლით რესურს მოითხოვს, ამიტომაც ჯობს ჯამები წინასწარ შევინახოთ სტრიქონების მასივში:

```
unsigned row_offsets[4] = { 0,  3,   4 };
unsigned columns[5] = { 0,   3,   5,   0,   3 };
double   values[5] = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

ამ ფორმატში: row_offsets[i] == sum(row_counts[0 : i - 1]). ამ შემთხვევაში (2, 3) ელემენტის მოძებნა გულისხმობს მხოლოდ პირდაპირ row_offsets[2]-ის ამოკითხვას ჯამის ოპერაციების შესრულების გარეშე. ზუსტად ეს საბოლოო ვარიანტი არის CSR. ამ ფორმატში COO-სგან განსხვავებით მეხსიერება და ძებნის დრო უფრო იზოგება. სტანდარტულ ბიბლიოთეკებში არსებული CSR ფორმატი სტრიქონების მასივის ბოლო ელემენტად ინახავს არანულოვანი ელემენტების რაოდენობას (რაც ხშირ შემთხვევებში საჭიროა): row_offsets[m+1] = nnz.

BSR (Block Sparse Row) ფორმატი

მსგავსად CSR-ისა, რომელიც იყო COO ფორმატის გაგრძელება, BSR ფორმატიც არის CSR-ის გაუმჯობესება/განზოგადება. BSR ცდილობს დააჩქაროს მატრიცებთან დამოკიდებული გამოთვლითი სისწრაფე რაღაც მოცულობის მეხსიერების სანაცვლოდ.

მაგალითად, მოცემული გვაქვს შემდეგი მატრიცი:

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{4} & \mathbf{5} \\ 0 & 0 & \mathbf{3} & \mathbf{6} \end{bmatrix}$$

ჩვენ თავისუფლად შეგვიძლია ამ მატრიცის მკვრივ, COO, ან CSR ფორმატში შენახვა. თუმცა, მატრიცზე დაკვირვებიდან შეიძლება დავინახოთ, რომ მატრიცს გააჩნია რაღაცა სტრუქტურა. კერძოდ, მატრში ვხედავთ ორ 2x2 ზომის ნულოვან ქვემატრიცებს: ერთი, რომელიც იწყება (0, 0) ინდექსზე და მეორე, რომელიც იწყება (2, 2)-ზე. საზოგადოდ, გამოდის, რომ მრავალ მეჩხერ მატრიცში ასეთ სტრუქტურას ხშირად ვხვდებით. BSR არის ზუსტად ის სტრუქტურა, რომელიც გათვლილია ასეთი „ბლოკურობასთან“ სამუშაოდ.

BSR ფორმატის თავდაპირველი ნაბიჯია მატრიცის BxB ზომის ბლოკებად დაყოფა. ჩვენი მაგალითისთვის, ერთეული ბლოკის ზომა არის 2x2 (B=2). ვიღებთ შემდეგ დანაწევრებულ ქვემატრიცებს:

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} \\ \mathbf{1} & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 \\ 0 & \mathbf{4} \end{bmatrix} \quad \begin{bmatrix} \mathbf{5} \\ \mathbf{3} & \mathbf{6} \end{bmatrix}$$

[0 0] [**4 5**]

[0 0] [**3 6**]

ჩვენ შეგვიძლია წარმოვიდგინოთ მიღებული მატრიცები, როგორც 2x2 ზომის მატრიცების მატრიცი. BSR იღებს ამ მატრიცების მატრიცს და ინახავს მას CSR ფორმატში. მსგავსად CSR ფორმატისა აქაც ნულების შემცველი ქვემატრიცი არ ინახება:

```
unsigned block_size = 2;
unsigned block_row_offsets[3] = { 0, 1, 2 };
unsigned block_columns[2] = { 0, 1 };
```

ეს მასივები გვიჩვენებს, რომ გვაქვს ორი არანულოვანი ბლოკი. block_size გვატყობინებს რომ ერთეული ბლოკის სიმაღლე და სიგანე არის 2.

მნიშვნელობების ბლოკი BSR-ში ინახავს მთლიან ბლოკს ნულებიანად, რადგანაც ნულების შენახვა აუცილებელია ყველა ბლოკის ზომის შესანარჩუნებლად:

```
unsigned block_size = 2;
unsigned block_row_offsets[3] = { 0, 1, 2 };
unsigned block_columns[2]     = { 0, 1 };

double values[2 * block_size * block_size] = {
       1.0, 2.0, 1.0, 0.0,  // block (0, 0) in row-major order
       4.0, 5.0, 3.0, 6.0 } // block (1, 1) in row-major order
```

თუ გვაინტერესებს ორიგინალური მატრიცის (2, 3) ელემენტი, მაშინ მოგვიწევს ორი ეტაპის შესრულება: პირველად ვიპოვით ბლოკს, რომელსაც (2, 3) ელემენტი ეკუთვნის და შემდეგ, ნაპოვნ ბლოკში ვიპოვით შიდა-ბლოკურ დაშორებას (intr-block offset), რომელიც შეიცავს (2, 3) ელემენტს. თუ ბლოკი არ არსებობს, მაშინ მნიშვნელობა უნდა იყოს 0.

ბლოკის პოვნა მარტივია, უბრალოდ შევასრულებთ მთელ გაყოფას სამიზნე კოორდინატებისა ბლოკის ზომაზე. ჩვენ მაგალითში ვიღებთ: (2, 3) / (2, 2) => (1, 1).

შემდეგ შეგვიძლია გადავინაცვლოთ (1, 1) ბლოკში. ვხვდებით, რომ (1, 1) ბლოკის მნიშვნელობები იწყება მნიშვნელობების მასივის (1 * block_size * block_size) დაშორებიდან. შემდეგ ჩვენ ვიგებთ შიდა-ბლოკურ კოორდიანტს, (0, 1)-ს შემდეგნაირად:

საძიებო კოორდინატს, (2, 3)-ს გამოკლებული ბლოკის საწყისი კოორდინატი, (2, 2)-ი.

საბოლოოდ, შიდა-ბლოკური დაშორებისა (0 * block_size + 1) და მთლიანი ბლოკის დაშორების (1 * block_size * block_size) კომბინაციით ვიღებთ დაშორებას: block_size * block_size + 1 = 5. და საძიებო ელემენტი არის values[5], რომელიც შეიცავს 5.0-ს.

BSR ფორმატში ძებნა შედარებით რთულად გაგები და დასაიმპლემენტირებელია, თუმცა მაინც უფრო ეფექტურია ვიდრე COO ფორმატი.

DIA (Diagonal) ფორმატი

ჩვენ განვიხილეთ რამდენიმე მეჩხერი მატრიცის ფორმატი: უმარტივესი მკვრივი (dense) ფორმატი მონაცემთა მიმართ ყველაზე აგნოსტურია (data-agnostic). მას არავითარი წინასწარი ვარაუდი არ აქვს იმის შესახებ თუ რას ინახავს და უბრალოდ ინახავს ყველაფერს. COO და CSR არიან შედარებით ნაკლებად აგნოსტურები მონაცემთა მიმართ; ამ ორ ფორმატში შენახული მონაცემი უნდა იყოს მეჩხერი, და თუ არ არის, მაშინ მეხსიერება ვერ დაიზოგება. BSR-მა ამოცანა კიდევ უფრო შეზღუდა და მოითხოვა, რომ მატრიცები „ბლოკურად მეჩხერები“ ყოფილიყვნენ.

ახლა განვიხილავთ ყველაზე შეზღუდულ ფორმატს, DIA-ს (ანუ Diagonal-ს, დიაგონალურს). DIA აქცენტს აკეთებს ისეთი მეჩხერი მატრიცების მეხსიერების შემცირებასა და დაჩქარებაზე, რომლებიც „მძიმედ დიაგონალურია“. მძიმედ დიაგონალურობა გულისხმობს, რომ არანულოვანი ელემენტები მთავარ დიაგონალთანაა განლაგებული:

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} & 0 & 0 \\ \mathbf{3} & \mathbf{1} & \mathbf{2} & 0 \\ 0 & \mathbf{3} & \mathbf{1} & \mathbf{2} \\ 0 & 0 & \mathbf{3} & \mathbf{1} \end{bmatrix}$$

მოკლედ რომ ვთქვათ, DIA იღებს საწყისს მატრიცს და ქმნის ახალ, ტრანსფორმირებულ მატრიცს, რომელიც სტრიქონებად ინახავს არსებული მატრიცის არანულოვან დიაგონალებს. ზედა მატრიცს აქვს სამი არანულოვანი დიაგონალი:

1. ერთი დიაგონალი, რომელიც იწყება (0, 0) ინდექსიდან და მთავრდება (3, 3) ინდექსზე, და შეიცავს ოთხ ერთიანს.
2. დიაგონალი, რომელიც იწყება (0, 1)-ზე, მთავრდება (2, 3)-ზე და შეიცავს სამ ორიანს.
3. დიაგონალი, რომელიც იწყება (1, 0)-ზე, მთავრდება (3, 2)-ზე და შეიცავს სამ სამიანს.

DIA-ში შესანახად, არსებული მატრიცი გარდაიქმნება ახალ მატრიცად 3 სტრიქონითა (3 არანულოვანი დიაგონალისთვის) და 4 სვეტით (მაქსიმალური დიაგონალის ზომისთვის):

$$\begin{bmatrix} 0 & \mathbf{3} & \mathbf{3} & \mathbf{3} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{2} & \mathbf{2} & \mathbf{2} & 0 \end{bmatrix}$$

ახალ მატრიცში სიცარიელე ივსება ნულებით. არსებული მატრიცის გარდა, დამატებით საჭიროა diagonal_ids მასივი, რომელიც შეიცავს უნიკალურ იდენტიფიკატორებს ყოველი არანულოვანი დიაგონალისთვის. დიაგონალების იდენტიფიკატორები და დიაგონალების მატრიცი საკმარისია საწყისი მეჩხერი მატრიცის ასაგებას:

```
unsigned max_diagonal_length = min(nrows, ncols);
unsigned diagonal_ids[3] = { -1, 0, 1 };
double values = {
              0.0, 3.0, 3.0, 3.0,
              1.0, 1.0, 1.0, 1.0,
              2.0, 2.0, 2.0, 0.0 }
```

diagonal_ids[i] შეიცავს დიაგონალების მატრიცში i-ური დიაგონალის იდენტიფიკატორს. საწყისს, (0, 0) პოზიციაზე მდგომ დიაგონალს (მთავარ დიაგონალს) ენიჭება იდენტიფიკატორი 1. მთავარი დიაგონალის ქვემოთ მყოფებს ენიჭებათ უარყოფითი რიცხვი, რომელიც იზრდება იმდენით რამდენითად სცილდება მთავარ დიაგონალს. მაღლა მყოფებზე მოქმედებს მსგავსი პრინციპი, ოღონდ მათ ენიჭებათ დადებითი იდენტიფიკატორები. ამ იდენტიფიკატორებით მარტივი გამოსათვლელია, თუ რომელ დიაგონალს ეკუთვნის მოცემული კოორდინატები. ელემენტი (r, c) ეკუთვნის (c - r) დიაგონალს. მაგალითად, ელემენტი (2, 1) საწყის მატრიცში ეკუთვნის დიაგონალს (c - r) = 1-2 = -1.

(r, c) ელემენტის მოძებნა DIA ფორმატში მარტივია. პირველად ვპოულობთ დიაგონალს იდენტიფიკატორის მიხედვით. თუ ასეთი დიაგონალი ვიპოვეთ და არის diagonal_ids[i]-ში, მაშინ მნიშვნელობა (r, c), ორიგინალურ მატრიცში, არის მოთავსებული მნიშვნელობებში სტრიქონ i-ზე და სვეტ r-ზე.

ჩვენ უკვე აღვნიშნეთ, რომ აღწერილ ფორმატებს შორის DIA ყველაზე მოუქნელი ფორმატია. დავუბრუნდეთ ჩვენ გიგანტურ, 1,000,000 x 1,000,000 ელემენტიან მატრიცს და ვთქვათ, რომ (0, 999999)-ზე მყოფი ჩანაწერი არანულოვანია. ზედა მარჯვენა კუთხეში მჯდომი (0, 999999) ელემენტი არის ერთელემენტიანი დიაგონალი და, რადგანაც ჩვენ ვინახავთ მონაცემებს DIA ფორმატში, ჩვენ გვიწევს 1,000,000 ჩანაწერის შენახვა ყველა დიაგონალისთვის, მიუხედავად იმისა თუ რა სიგრძისაა თითოეული მათგანი. ამით ჩვენ ვხარჯავთ 999,999 * 8 ბაიტს მაშინ, როდესაც იმ ერთ ელემენტს COO ფორმატის გამოყენებით შევინახავდით მხოლოდ 16 ბაიტში.

მიუხედავად ზემოთქმულისა ძლიერად დიაგონალური მატრიცის შემთხვევაში DIA სხვა ნახსენებ ფორმატებზე ეფექტურია სიჩქარისა და მეხსიერების მხრივ.

ELL (Ellpack-Itpack) ფორმატი

ELL ფორმატი მსგავსია CSR, COO, და BSR ფორმატებისა. მარტივად რომ აღვწეროთ, ELL ფორმატი აპატარავებს არსებულ მატრიცს სვეტების რაოდენობის შემცირებით. განვიხილოთ შემდეგი მატრიცი:

[**1** **2** **3** 0]

[0 **4** **5** 0]

[0 **6** 0 **7**]

[**8** 0 0 0]

არსებული მატრიცის ELL ფორმატში გადაყვანა იწყება იმ სტრიქონის ძებნით, რომელიც შეიცავს ყველაზე მეტ არანულოვან ელემენტს (ჩვენ შემთხვევაში ეს სტრიქონია 0 და შეიცავს 3 არანულოვან ელემენტს). ELL ქმნის ორ მასივს არსებული მატრიცის შესანახად. იქმნება მნიშვნელობების მასივი განზომილებით: (nrows) x (max_nnz_per_row), სადაც max_nnz_per_row არის წინა ბიჯში ნაპოვნი მნიშვნელობა (ჩვენ შემთხვევაში 3). სვეტების მასივი იგივე განზომილებისაა, რაც მნიშვნელობათა მასივი (მატრიცი). ELL ყველა სტრიქონს აშორებს ნულებს, მარცხნივ წევს მათ და დარჩენილ ადგილს ავსებს ნულებით. სვეტების მასივი შეიცავს ელემენტის სვეტის ინდექსს ორიგინალურ მასივში. შედეგად ზედა მატრიცისთვის ვიღებთ შემდეგ ორ მატრიცსს:

Values

[**1** **2** **3**]
[**4** **5** 0]
[**6** **7** 0]
[**8** 0 0]

Columns

[0 1 2]
[1 2 -1]
[1 3 -1]
[0 -1 -1]

ელემენტის მოძებნა მარტივად ხდება: იპოვე შესაბამისი სტრიქონი სვეტების მასივში და იპოვე სვეტის ნომერი. თუ იპოვე, მაშინ მნიშვნელობების მასივში ელემენტის მნიშვნელობა არსებულ i-ურ და j-ურ პოზიციაზე იქნება, თუ ვერ იპოვე ესეიგი ელმენტი ნულია.

შორიდან ჩანს თითქოს ELL ფორმატი არაეფექტურია და არ იძლევა მეხსიერების ეკონომიას ნულების შენახვისა და დამატებითი სვეტების მასივის გამო, მაგრამ გააჩნია რა ამოცანაში და რა გარემოში, როგორ არქიტექტურაში ხდება მისი გამოყენება.

უმეტესობა ფორმატებისა აბსტრაქტული და ენისგან დამოუკიდებელია, რაც ნიშნავს იმას, რომ მათი იმპლემენტაცია ბევს სხვადასხვა პროგრამირების ენებშია შესაძლებელი, მაგრამ არსებობს პროგრამირების ენებზე დამოკიდებული მეჩხერი მატრიცის ფორმატებიც. ერთ-ერთია, მაგალითად, JSA (Java Sparse Array) ფორმატი, რომელიც მორგებულია კონკრეტულად Java პროგრამირების ენაზე.

JSA (Java Sparse Array) ფორმატი

JSA ფორმატი იმპელემეტირებულია Java პროგრამირების ენაში და იგი წარმოადგენს ELL ფორმატის ეფექტურ განზოგადებას. Java-ში, ისევე როგორც C ენის საფუძველზე შექმნილ სხვა ენებში, ორგანზომილებიანი მასივი წარმოადგენს ერთგანზომილებიან მასივს, რომლის ელემენტები ასევე ერთგანზომილებიანი მასივებია.

Java-ში ყოველ ერთგანზომილებიან მასივს „კარგად ახსოვს“ თავისი ზომა, ამიტომ საკმარისია საწყისი მეჩხერი მართკუთხა მასივის ნაცვლად განვიხილოთ ორგანზომილებიანი მასივი, რომელიც მიიღება საწყისი მატრიციდან ნულების ამოყრით. ცხადია, ამ მატრიცის სტრიქონები სხვადასხვა სიგრძისაა, მაგრამ Java-ში, როგორც აღვნიშნეთ, ეს არ წარმოადგენს პრობლემას. საჭიროა აგრეთვე ინდექსების მატრიცის შენახვა, რომელშიც ჩაიწერება არანულოვანი ელემენტების სვეტების ინდექსები.

[7]-ში, ასევე ამ ამ სტატიის ავტორების სხვა ნაშრომებში, ჩატარებულია გარკვეული ტესტები, რომლებიც ამტკიცებენ მათ მიერ შემუშავებული ფორმატის („ჯავას მეჩხერი მასივი“, Java Sparse Array) ეფექტურობას. შევნიშნოთ, რომ ამ ორი მატრიცის გარდა, საჭიროა აგრეთვე მასივების სტრიქონების რაოდენობის შენახვა.

მაგალითად, შემდეგი მატრიცი:

$$\begin{bmatrix} \mathbf{1} & 0 & \mathbf{2} & 0 & 0 \\ \mathbf{3} & \mathbf{4} & 0 & \mathbf{5} & 0 \\ 0 & \mathbf{6} & \mathbf{7} & 0 & \mathbf{8} \\ 0 & 0 & \mathbf{9} & \mathbf{10} & 0 \\ 0 & 0 & \mathbf{2} & \mathbf{11} & \mathbf{12} \end{bmatrix}$$

ჯავას მეჩხერი მასივის ფორმატში ჩაიწერება შემდეგი სახით:

```
double[][] value = { { 1, 2 }, { 3, 4, 5 }, { 6, 7, 8 }, { 9, 10 }, { 11, 12 } };
int[][] index = { { 0, 2 }, { 0, 1, 3 }, { 1, 2, 4 }, { 2, 3 }, { 3, 4 } };
```

ანუ value[1] = (3, 4, 5), value[2][1] = 7, index[3][0] = 2 და ა. შ.

[7]-ის მიხედვით, ამ ფორმატის გამოყენების შემთხვევაში შესანახი არის 2 * nnz + 2 * n ელემენტი, განსხვავებით CSR ფორმატის შემთხვევაში საჭირო 2 * nnz + n + 1 ელემენტისგან. თუმცა, ეს შეფასება შემსუბუქებულია. იგი სამართლიანი იქნებოდა, ჯავას მასივები რომ წარმოადგენდენ პოინტერებს და არა გარკვეული კლასის ობიექტებს, რომლებმაც ობიექტის მისამართთან ერთად უნდა შეინახონ გარკვეული დამატებითი ინფორმაცია.

JNZ (Jagged Non-zero) ფორმატი

ზემოთ ადღწერილი JSA ფორმატის C++ პროგრამირების ენაში მოწყობა სავსებით შესაძლებელია, მაგრამ ეს არ იქნება საუკეთესო არჩევანი. JNZ ფორმატში ჩვენ ვქმნით ოდნავ განსხვავებული ფორმის კბილებიანი მატრიცების წყვილს. მისი იმპლემენტირება შესაძლებელია ნებისმიერ ენაში, რომელსაც შეუძლია პოინტერებთან ოპერირება.

განსხვავება ეხება მხოლოდ მთელრიცხვა, ინდექსების მატრიცას. მას ჩვენ ვამატებთ ერთ სტრიქონს, რომლის პირველი ელემენტი არის საწყისი მატრიცის სტრიქონების რაოდენობა, ხოლო დანარჩენ ელემენტებს წარმოადგენენ საწყისი მატრიცის სტრიქონებში არანულოვანი ელემენტების რაოდენობები. შედეგად, საწყისი მატრიცის i-ური სტრიქონის არანულოვანი ელემენტების სვეტების ინდექსები მოთავსებულია ინდექსების მატრიცის (i + 1)-ე სტრიქონში, რაც არ იწვევს არავითარ პრობლემას შესრულების დროის თვალსაზრისით.

კვლავ იგივე მატრიცის მაგალითზე, ეს მატრიცები ვიზუალურად შეგვიძლია წარმოვიდგინოთ შემდეგი სახით:

$$value = \begin{cases} (1. \quad 2.) \\ (3. \quad 4. \quad 5.) \\ (6. \quad 7. \quad 8.) \\ (9. \quad 10.) \\ (11. \quad 12.) \end{cases} \qquad index = \begin{cases} (5 \quad 2 \quad 3 \quad 3 \quad 2 \quad 2) \\ (0 \quad 2) \\ (0 \quad 1 \quad 3) \\ (1 \quad 2 \quad 4) \\ (2 \quad 3) \\ (3 \quad 4) \end{cases}$$

ახლა, index[0][0] არის 5-ის ტოლი და გვიჩვენებს საწყისი მატრიცის სტრიქონების რაოდენობას. index[0][3] გვიჩვენებს index მატრიცის მეოთხე (ანუ value მატრიცის მესამე) სტრიქონში ელემენტების რაოდენობას და არის 3-ის ტოლი. ცხადია, ეს ორი კბილებიანი მატრიცა იქმნება დინამიკურად ორმაგი პოინტერების გამოყენებით.

JNZ ფორმატის გამოყენების შემთხვევაში შესანახი არის 2 * nnz + 3 * n + 4 ელემენტი, აქედან მხოლოდ nnz რაოდენობა არის ნამდვილი რიცხვი. 2 * n + 1 არის პოინტერი (ორივე მატრიცის სტრიქონების შესაბამისი), 2 ორმაგი პოინტერი (მატრიცებისთვის), დანარჩენი კი მთელი რიცხვები.

როგორც ვხედავთ, თუ რომელიმე ფუნქციაში გადავაწვდით index და value პოინტერებს, რომლებიც უკვე მიუთითებენ რეალურად გამოყოფილ და შევსებულ მეხსიერების ფრაგმენტებს, მაშინ ამ ფუნქციიდან ადვილად აღვადგენთ (index[0]-ის საშუალებით) ყველა საჭირო ცნობას სტრიქონების რაოდენობის და თითოეულ სტრიქონში ელემენტების რაოდენობების შესახებ. უფრო მეტი, ამ ფორმატის გამოყენების შემთხვევაში ჩვენ ადვილად ვაპროგრამებთ მეჩხერი მატრიცის ვექტორზე გამრავლების ოპერაციას განმეორების შეტყობინებაში მკვეთრად შემცირებული შედარებებით.

მოვიყვანოთ JNZ-ქვემატრიცის ფორმატში წარმოდგენილი მეჩხერი მატრიცის ვექტორზე გამრავლების ფუნქციის კოდი, რომელიც გვარწმუნებს ორ ფაქტში: სწრაფი გამრავლების

ტრადიციული ტექნიკა ადვილი დასაპროგრამებელია და index კბილებიან მატრიცაში ზედმეტი სტრიქონის დამატება არანაირად არ აისახება სისწრაფეზე ან სირთულეზე:

```
void MatrixByVector(double **m, int **index, double *x, double* res)
{
	const int k(index[0][0]);
	int i, j, n5;
	double *p;
	int *q;
	int size;
	double result;

	for (i = 0; i < k; ++i)
	{
		result = 0.;
		p = m[i];
		q = index[i + 1];
		size = index[0][i + 1];
		n5 = size % 5;
		for (j = 0; j < n5; ++j)
			result += p[j] * x[q[j]];
		for (; j < size; j += 5)
		{
			result += p[j] * x[q[j]] + p[j + 1] * x[q[j + 1]]
				+ p[j + 2] * x[q[j + 2]] + p[j + 3] * x[q[j + 3]]
				+ p[j + 4] * x[q[j + 4]];
		}
		res[i] = result;
	}
}
```

ლოკალური ცვლადებიდან, m გამიზნულია მეჩხერი მატრიცის არანულოვანი ელემენტების შესაბამისი დინამიკური (კბილებიანი) ორგანზომილებიანი მასივის პოინტერის მისაღებად, index გამიზნულია ინდექსების მთელრიცხვა მატრიცის პოინტერის მისაღებად, x არის იმ ვექტორის პოინტერი, რომელზეც ვამრავლებთ და შედეგის პოინტერი არის res. პირველ რიგში, არ გვჭირდება სტრიქონების რაოდენობა, მისი ამოღება ხდება k ცვლადში. დანარჩენი ოპერაციები საკმაოდ მარტივია და აღარ განვმარტავთ.

საკმარისია აღინიშნოს, რომ ერთადერთი განსხვავება მკვრივი მატრიცის და ვექტორის გამრავლებისგან (გართულების მიმართულებით) არის x ვექტორში ინდექსის გამოთვლის აუცილებლობა. სამაგიეროდ, ფუნქციაში არგუმენტების რაოდენობა არ შეცვლილა (დაემატა ინდექსების რაოდენობა მაგრამ დააკლდა სტრიქონების რაოდენობა), და კოდის სტრუქტურა იდენტურია.

[6]-ში, ტესტების პირველ ჯგუფში, სადაც ჩვენ მეჩხერი მატრიცების ფორმატების ეფექტიანობას ვიკვლევთ CG (Conjugate Gradeint) ალგორითმისთვის, ვიყენებთ JNZ ფორმატის ვარიანტს, რომელიც ინახავს ინფორმაციას მხოლოდ დიაგონალისა და ზედა

სამკუთხა მატრიცის შესახებ. ამის ხარჯზე იზოგება მატრიცის შესანახი მეხსიერების ნახევარი. სამაგიეროდ შედარებით რთულდება განხილული ფუნქციის ანალოგის კოდი და იგი 15 - 25%-ით ნელი ხდება.

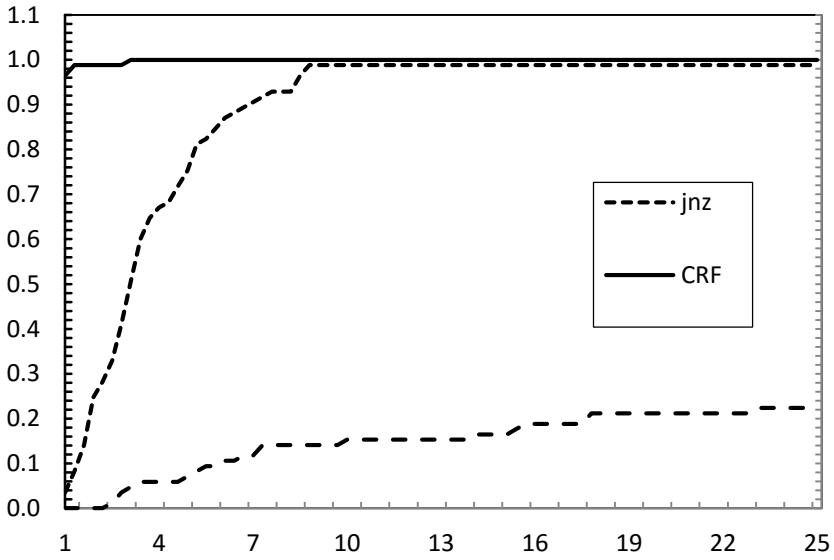
JNZ vs. CSR vs. Mapped Matrix

ყოველი სიმეტრიული, დადებითად განსაზღვრული მეჩხერი მატრიცა და შესაბამისი განზომილების მქონე ვექტორი ცალსახად განსაზღვრავს Ax=b სახის სისტემას, რომელიც ამოხსნადია CG (Conjugate Gradient) მეთოდით. სატესტო ამოცანების სიმრავლე ჩვენს ([6]-ის) ექსპერიმენტებში შედგება დაახლოებით 90 ამოცანისგან (მატრიცა და ვექტორი-მარჯვენა მხარე) - სიმეტრიული, დადებითად განსაზღვრული მეჩხერი მატრიცა და შესაბამისი განზომილების მქონე ვექტორი. 85 ასეთი მატრიცა აღებულია [5]-იდან. ისინი პირობითად შეგვიძლია გავყოთ სამ ჯგუფად: მცირე, საშუალო და დიდი ზომის. ყოველი მატრიცისთვის შემთხვევითად არის გენერირებული შესაბამისი განზომილების ვექტორი, რომელიც ყოველთვის განიხილება ამ მატრიცასთან ერთად.

Ax=b სისტემის ამოსახსნელად CG მეთოდთან ერთად საჭირო არის მეჩხერი მატრიცის წარმოდგენის რომელიმე ფორმატის განხილვა. მეთოდების სიმრავლის ფორმირებისთვის, CG მეთოდს და მატრიცის წარმოდგენის კონკრეტულ ფორმატს ჩვენ განვიხილავთ როგორც ცალკე აღებულ მეთოდს. რადგან ამომხსნელი (solver) დაფიქსირებულია, ჩვენ პრაქტიკულად მხოლოდ მონაცემთა სტრუქტურების შედარებას ვახდენთ.

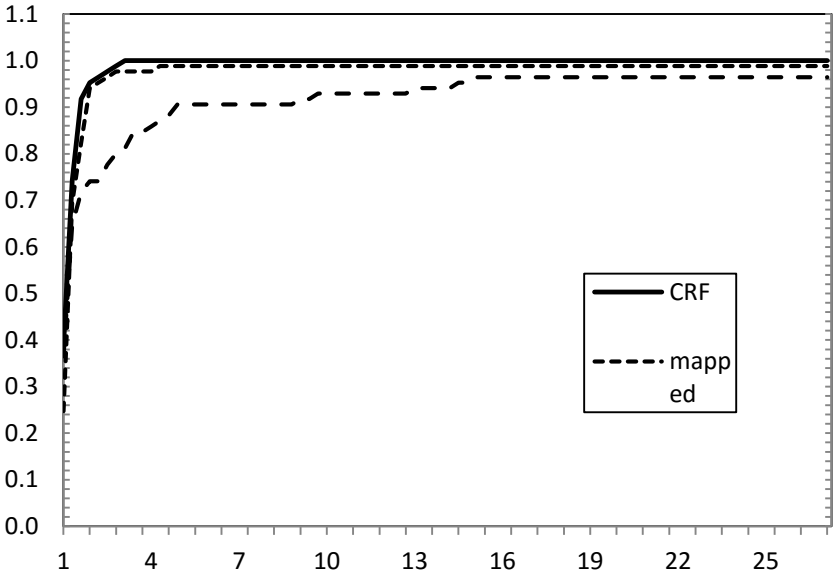
ექსპერიმენტების შედეგებში მონაწილეობს სამი ფორმატი. ერთი არის JNZ ფორმატი, რომელიც ჩვენ მიერ არის იმპლემენტირებული რამდენიმე ალგებრულ ალგორითმთან ერთად (რაც საჭიროა CG მეთოდში მისი ინტეგრაციისთვის), ორი სხვა არის Boost ბიბლიოთეკის Mapped Matrix და Compressed Matrix ფორმატები.

ტესტირების შედეგები ასახულია შემდეგ სამ პროფაილში. პირველი მათგანი აგებულია შერჩეული ამოცანების ამოხსნისთვის საჭირო ჯამური დროების გათვალისწინებით (მატრიცის შევსებას დამატებული სისტემის ამოხსნის დრო):



ფიგურა 1 Performance profile for full times

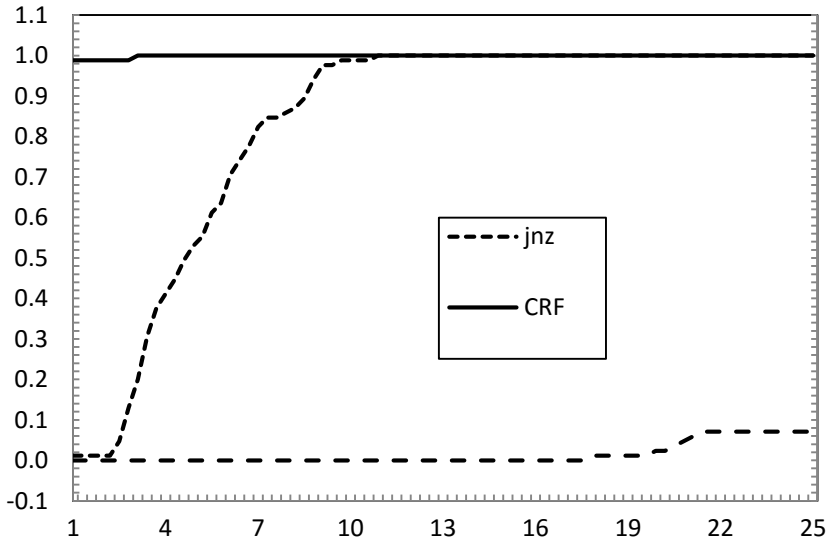
მეორე პროფაილი აგებულია განხილულ ამოცანებში ფაილიდან მატრიცების შევსების დროების მიხედვით. როგორც ცნობილია, დიდი ზომის მატრიცებისთვის ფაილიდან მონაცემების წაკითხვა შრომატევადი და ხანგრძლივი პროცედურაა, ამიტომ მისი განხორციელება არ არის ხელსაყრელი სტანდარტული მეთოდების გამოყენებით.



ფიგურა 2 Performance profile for filling times

ვხედავთ, რომ შედარებით ახალი ფორმატები ბევრად სწრაფად ივსება CSR ფორმატთან შედარებით.

ბოლოს, თუ მხოლოდ სისტემების ამოხსნის დროებს განვიხილავთ, გვექნება შემდეგი პროფაილი:



ფიგურა 3 Performance profile for solving times

მეჩხერი მატრიცების იმპლემენტაციები

არსებობს მრავალი, სხვადასხვა პროგრამირების ენაზე დაწერილი იმპლემენტაცია მეჩხერი მატრიცებისა (კერძოდ მათი ფორმატებისა), რომელთა უმრავლესობასაც ბიბლიოთეკის სახე აქვს მიცემული. ზოგიერთი ასეთი ბიბლიოთეკა, გარდა კონტეინერისა, შეიცავს ე. წ. ამომხსნელებს (solver), რომლებიც გამოიყენება მეჩხერი მატრიცების ოპერაციებში (ვექტორზე ნამრავლი, მატრიცზე ნამრავლი და ა. შ.).

C++ ენის ყველაზე ცნობილი ბიბლიოთეკებია: Boost Libraries - uBLAS, GSL, ALGLIB, PETSc, Eigen3; Fortran-სთვის: MUMPS; ხოლო ორივესთვის: PaStix და SuperLU.

ამ იმპლემენტაციებიდან ვერ ვიტყვით, რომ ყველა შეიცავს ყველა ფორმატის იმპლემენტაციას: ზოგი შეიცავს სხვაზე მეტს - ზოგი კი პირიქით ნაკლებს. ის ფორმატები, რომელბიც ყველაზე პოპულარულია გვხვდება თითქმის ყველა იმპლემენტაციაში (მაგალითად, CSR ფორმატი, CSC ფორმატი და ა. შ.).

მაგალითისთვის შეგვიძლია განვიხილოთ Boost Libraries-ში შემავალი uBLAS (Basic Linear Algebra Subprograms) ბიბლიოთეკა. საზოგადოდ, Boost Libraries ეს არის ბიბლიოთეკების ნაკრები, რომელიც შეიცავს სხვადასხვა ბიბლიოთეკებს - სხვადასხვა კატეგორიის პრობლემია თუ ამოცანის გადასაჭრელად.

uBLAS არის Boost-ის ერთ-ერთი ბიბლიოთეკა, რომელიც გვთავაზობს მატრიცებსა და ვექტორებს მკვრივ და მეჩხერ მონაცემებთან სამუშაოდ. მასში იმპლემენტირებულია CSR, CSC, COO და Mapped Matrix (DOK) ფორმატები. ბიბლიოთეკა იყენებს C++-ში მხარდაჭერილ შაბლონებს (Template Class Library) და ამით უფრო მოქნილ და მოხერხებულ ინტერფეისს გვთავაზობს. ბიბლიოთეკა დიდი ხანია რაც არსებობს, ჰყავს ბევრი კონტრიბუტორი და მასში ხორციელდება სისტემატური განახლებები, ამიტომაც მიიჩნევა, რომ იგი უფრო სანდო და მდგრადია.

განვიხილოთ uBLAS-ის გამოყენებით CSR ფორმატის კონტეინერის აგება და შევსება:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main() {
	using namespace boost::numeric::ublas;
	compressed_matrix<double> m(3, 3, 3 * 3);
	for (unsigned i = 0; i < m.size1(); ++i) {
		for (unsigned j = 0; j < m.size2(); ++j) {
			m(i, j) = 3 * i + j;
		}
	}
	std::cout << m << std::endl;
}
```

გაჩუმებით იგულისხმება სტრიქონით დალაგება. შეგვიძლია შევცვალოთ და მიმთითებლის განსაზღვრებაში ტიპის შემდეგ დავუწეროთ `column_major`:

```
compressed_matrix<double, column_major> m(3, 3, 3 * 3);
```

შესაძლებელია ასევე წრფივი განტოლების ამოხსნა:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/lu.hpp>
#include <boost/numeric/ublas/io.hpp>

int main() {
	using namespace boost::numeric::ublas;
	compressed_matrix<double> A(5, 5, 12);

	A(0, 0) = 2.; A(0, 1) = 3.; A(1, 0) = 3.; A(1, 2) = 4.;
	A(1, 4) = 6.; A(2, 1) = -1.; A(2, 2) = -3.; A(2, 3) = 2.;
	A(3, 2) = 1.; A(4, 1) = 4.; A(4, 2) = 2.; A(4, 4) = 1.;

	vector<double> y(5);
	y(0) = 8.; y(1) = 45.; y(2) = -3.; y(3) = 3.; y(4) = 19.;

	boost::numeric::ublas::permutation_matrix<size_t> pm(A.size1());
	lu_factorize(A, pm);
	lu_substitute(A, pm, y);

	std::cout << y << std::endl; // output: [5] (1, 2, 3, 4, 5)
}
```

მეჩხერი მატრიცების კოლექციები

მეჩხერი მატრიცის პროგრამულად დაგენერირება რთულ საქმეს არ წარმოადგენს. მთავარია შეძლო შემთხვევითი რიცხვის გენერირება, მაგრამ ამოცანა შეიძლება არ მოითხოვდეს შემთხვევით რიცვს. შეიძლება იგი მოითხოვდეს რაიმე რეალური დაკვირვებებიდან მიღებულ შედეგებს ანდა მოითხოვდეს მატრიცს, რომელიც სიმეტრიულია, დადებითად განსაზღვრული, კვადრატული და ა. შ. ასეთ შემთხვევაში რთულია და ზოგჯერ შეუძლებელიც მსგავსი მონაცემების მოპოვება, ამიტომაც არსებობს წინასწარ აგებული დიდი და მცირე ზომის, მეჩხერი და მკვრივი მატრიცები კატეგორიზებული გარკვეული მახასიათებლების მიხედვით სხვადასხვა კოლექციებში. ამ მატრიცებს იყენებენ სხვადასხვა ამოცანებში, როგორც სატესტო ნიმუშებს. არსებობს ბევრი კოლექცია, რომელიც იძლევა მსგავს მატრიცებს. კოლექცია შეიძლება წარმოადგენდეს ვებ-გვერდს, რომლიდანაც შეგიძლია მატრიცების გადმოწერა, ანდა აპლიკაციას, რომლებსაც სისტემაში აყენებ.

დღეს ბევრი ასეთი კოლექცია არსებობს:

- Harwell-Boeing Collection
- SPARSKIT Collection
- NEP Collection
- MMDELI
- Independent Sets and Generators
- Universal Java Matrix Package (UJMP)
- UFget: MATLAB and Java interface to the UF Sparse Matrix Collection

აქედან ყველაზე ცნობილი კოლექციაა Harwell-Boeing (მოკლედ, HB) კოლექცია. სხვა კოლექციებსა და მატრიცებზე ინფორმაციის ნახვა შესაძლებელია MatrixMarket-ის საიტზე: http://math.nist.gov/MatrixMarket/

მატრიცების ფაილური ფორმატები

კოლექციებში აღწერილი მატრიცები მოცემულია შესაბამის ASCII ფორმატებში. დღესდღეობით ცნობილია ორი ასეთი ფაილური ფორმატი მატრიცების შესანახად: Harwell-Boeding Exchange Format და Matrix Market Exchange Format. ამ ორიდან ყველაზე პოპულარულია Harwell-Boeding-ის ფორმატი მეჩხერი მატრიცების გაცვლისას. იგი აგრეთვე ყველაზე დეტალურია. მისი ფაილი, მსგავსად HTML ფაილისა, შედგება ორი ნაწილისაგან: თავისა და ტანისგან. თავი შეიცავს ამომწურავ ინფორმაციას მატრიცის შესახებ. მხოლოდ თავის ამოკითხვით მომხმარებელს შეუძლია დაადგინოს რა მოცულობა იქნება საჭირო მატრიცის შესანახად და სხვა დამატებითი ინფორმაცია.

განსხვავებით HB ფორმატისგან, Matrix Market ფორმატი საგრძნობლად მარტივია. ისიც მსგავსად HB ფორამტისა ორი ნაწილისგან შედგება, მაგრამ თავში ნაკლებ ინფორმაციას შეიცავს: მოკლე აღწერას, განზომილებასა და არანულოვანი ელემენტების რაოდენობას.

ხოლო ტანი შედგება (i, j, value) წყვილებისგან. თუ მატრიცი დიაგონალურია ან სიმეტრიული, ამ ფორმატში დუბლირებული, ზედმეტი ინფორმაცია ამოღებულია.

ორივე ფაილურ ფორმატს აქვს მზა ინტერფეისი ფაილიდან ინფორმაციის ამოსაკითხად და ფაილის შესაქმნელად დაწერილი C++-ისთვის, Fortran-ისთვის, Matlab-ისთვისა და Python-ისთვისაც კი. შეგვიძლია განვიხილოთ C++-ის კოდი .mm (Matrix Market) ფორმატის ფაილის წასაკითხად:

```c
#include <stdio.h>
#include <stdlib.h>
#include "mmio.h"

int main(int argc, char *argv[]) {
	int ret_code;
	MM_typecode matcode;
	FILE *f;
	int M, N, nz;
	int i, *I, *J;
	double *val;

	if ((f = fopen("matrix.mm", "r")) == NULL)
		exit(1);

	if (mm_read_banner(f, &matcode) != 0)
	{
		printf("Could not process Matrix Market banner.\n");
		exit(1);
	}

	/*  This is how one can screen matrix types if their application */
	/*  only supports a subset of the Matrix Market data types.      */
	if (mm_is_complex(matcode) && mm_is_matrix(matcode) &&
		mm_is_sparse(matcode)) {
		printf("Sorry, this application does not support ");
		printf("Market Market type: [%s]\n",
			mm_typecode_to_str(matcode));
		exit(1);
	}

	/* find out size of sparse matrix .... */
	if ((ret_code = mm_read_mtx_crd_size(f, &M, &N, &nz)) != 0)
		exit(1);

	/* reseve memory for matrices */
	I = (int *)malloc(nz * sizeof(int));
	J = (int *)malloc(nz * sizeof(int));
	val = (double *)malloc(nz * sizeof(double));

	/* NOTE: when reading in doubles, ANSI C requires the use of the "l"  */
	/*   specifier as in "%lg", "%lf", "%le", otherwise errors will occur */
	/*  (ANSI C X3.159-1989, Sec. 4.9.6.2, p. 136 lines 13-15)            */
	for (i = 0; i<nz; i++) {
		fscanf(f, "%d %d %lg\n", &I[i], &J[i], &val[i]);
		I[i]--;  /* adjust from 1-based to 0-based */
		J[i]--;
	}

	if (f != stdin) fclose(f);
```

```
        /************************/
        /* now write out matrix */
        /************************/
        mm_write_banner(stdout, matcode);
        mm_write_mtx_crd_size(stdout, M, N, nz);
        for (i = 0; i<nz; i++)
                fprintf(stdout, "%d %d %20.19g\n", I[i] + 1, J[i] + 1, val[i]);
}
```

გამოყენებული ლიტერატურა:

1. https://en.wikipedia.org/wiki/Sparse_matrix
2. https://medium.com/@jmaxg3/101-ways-to-store-a-sparse-matrix-c7f2bf15a229
3. https://pdfs.semanticscholar.org/cf79/2bb9ac27ea6b65aa31c091be1745555d0db9.pdf
4. Sparse Matrix File Formats - http://math.nist.gov/MatrixMarket/formats.html#hb
5. Sparse Matrix Collections - http://math.nist.gov/MatrixMarket/collections/hb.html
6. Jagged Non-zero Submatrix Data Structure \ https://www.sciencedirect.com/science/article/pii/S2346809217300727?via%3Dihub
7. G. Gundersen, T. Steihaug, Data structures in Java for matrix computations, Concurrency and Computation: Practice and Experience, 2004, 799-815
8. Yousef Saad - Iterative Methods for Sparse Linear Systems (second ed.), SIAM (2003)