ივანე ჯავახიშვილის სახელობის თბილისის
სახელმწიფო უნივერსიტეტი

ვახტანგ ლალუაშვილი

მეჩხერი მატრიცების წარმოდგენის ეფექტური ფორმატები და მათი ტესტირების საკითხი

ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერება

ხელმძღვანელი: პროფესორი კობა გელაშვილი

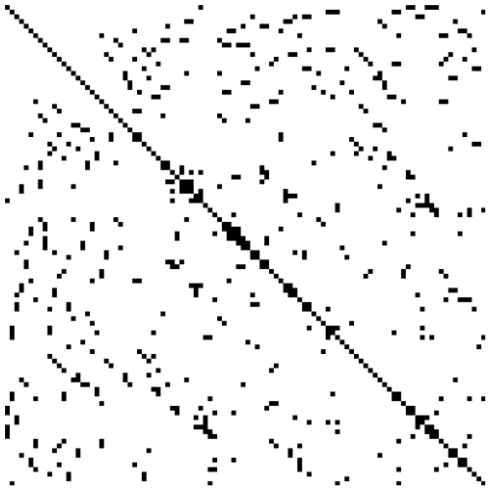
თბილისი, 2018

აბსტრაქტი

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.

[6]-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.

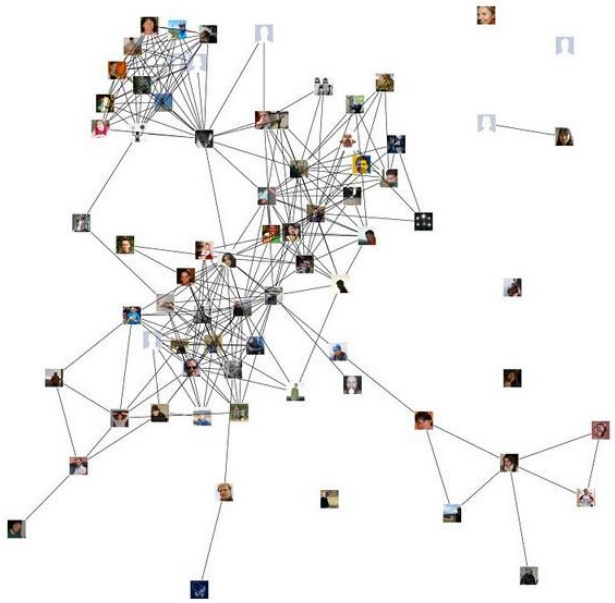
კოლოკვიუმში ჩვენ განვიხილავთ მეჩხერი მატრიცის ფორმატს სახელად JNZ (Jagged Non-zero) (იხ. [1]), რომელიც იმპლემენტირებულია C++ პროგრამირების ენაზე. აღგიწერთ სატესტო გარემოს და გიჩვენებთ ამ ფორმატის უპირატესობებს შესაბამისი ტესტების შედეგების საფუძველზე, რომელიც წინასწარ მომზადებულ გარემოში ამორჩეული მონაცემების გამოყენებით შემოწმდა. ჩვენ ვაგრძელებთ JNZ ფორმატის გაუმჯობესებაზე მუშაობას. ამიტომ ჩვენთვის მნიშვნელოვანია და საინტერესო [1]-ის შემფასებლების რეკომენდაციები. კოლოკვიუმის ბოლო ნაწილში ჩვენ განვიხილავთ სტატიის შემფასებლების რჩევებსა და შენიშვნებს ტესტებთან დაკავშირებით.



სურათი 1. მეჩხერი მატრიცის მაგალითი (შავი წერტილები არანულოვანი მონაცემებია)

მეჩხერი მატრიცები რეალურ ცხოვრებაში

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
- **SSK** – Symmetric Skyline
- **NSK** – Nonsymmetric Skyline
- **JAD** – Jagged Diagonal (or JDS)
- **SELL** – Sliced ELL

- **CSC** – Compressed Sparse Column
- **MSR** – Modified CSR
- **LIL** – Lined List (list of lists)
- **ELL** – Ellpack-Itpack
- **DIA** – Diagonal
- **BSR** – Block Sparse Row
- **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, 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.

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.

[10]-ში, ასევე 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.

[10]-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:

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

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.

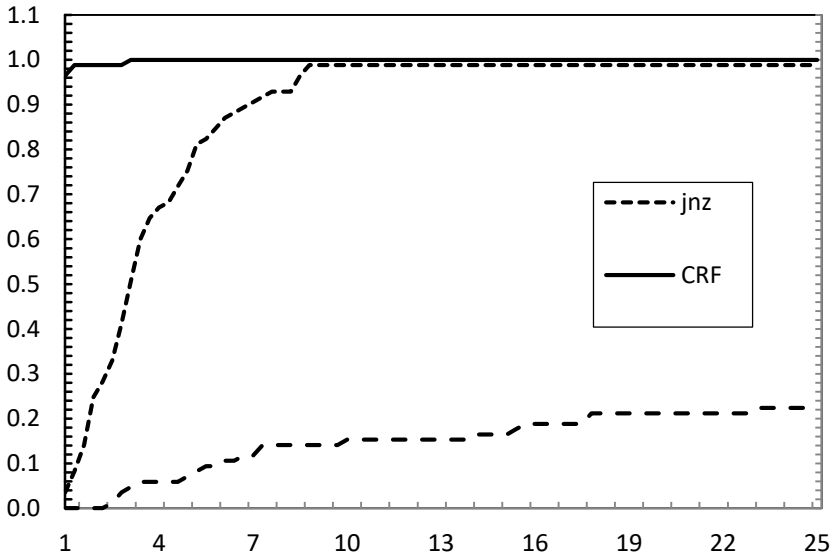
JNZ, CSR და Mapped Matrix-ის სატესტო გარემოში შემოწმება

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 [4]-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.

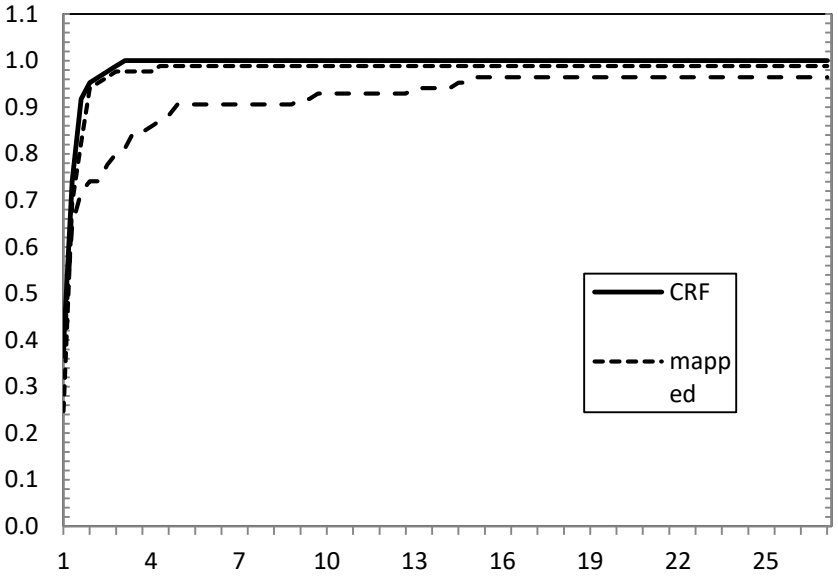
ექსპერიმენტების შედეგებში მონაწილეობს სამი ფორმატი. ერთი არის JNZ ფორმატი, რომელიც ჩვენ მიერ არის იმპლემენტირებული რამდენიმე ალგებრულ ალგორითმთან ერთად (რაც საჭიროა CG მეთოდში მისი ინტეგრაციისთვის), ორი სხვა არის Boost ბიბლიოთეკის Mapped Matrix და Compressed Matrix ფორმატები.

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

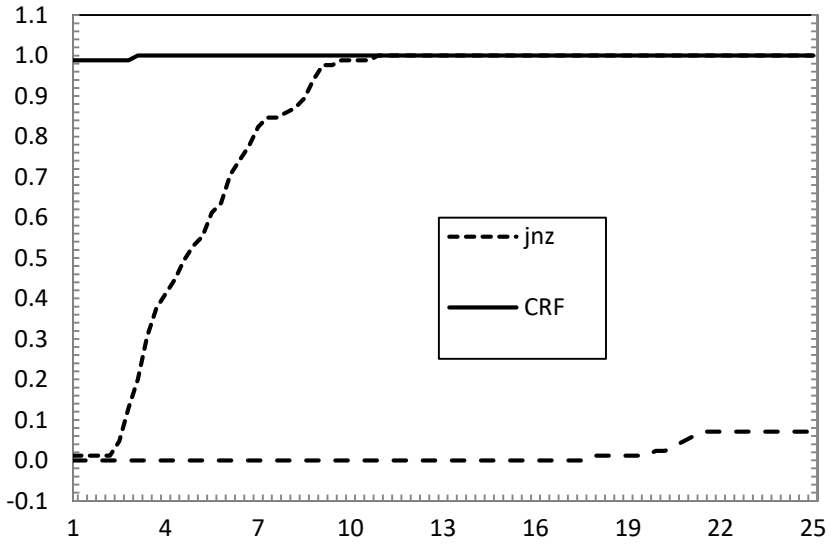
მეორე პროფაილი აგებულია განხილულ ამოცანებში ფაილიდან მატრიცების შევსების დროების მიხედვით. როგორც ცნობილია, დიდი ზომის მატრიცებისთვის ფაილიდან მონაცემების წაკითხვა შრომატევადი და ხანგრძლივი პროცედურაა, ამიტომ მისი განხორციელება არ არის ხელსაყრელი სტანდარტული მეთოდების გამოყენებით.



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.

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.

გამოქვეყნებული სტატია და შემფასებლების რჩევები

როგორც აქამდე ვახსენეთ, ჩვენი JNZ მეჩხერი ფორმატის შესახებ გამოქვეყნდა სტატია ანდრია რაზმაძის სახელობის მათემატიკური უნივერსიტეტის ჟურნალში [6]. სტატიის გამოქვეყნების პროცესში შემფასებლებმა შენიშვნები გაგვიკეთეს და რჩევები მოგვცეს. კერძოდ, გვირჩიეს განგვეხილა პრეკონდიცირება და არ გვეჩვენა მხოლოდ პირდაპირი შეუღლებული გრადიენტების მეთოდის მაგალითი. აგრეთვე, შენიშნეს რომ არ გვქონდა განხილული პარალელური ალგორითმები და ზოგადად, მრავალნაკადიანი გარემო. ჩვენ მივიღეთ შენიშვნები და დავიწყეთ ამ ორ საკითხზე მუშაობა.

მოდით ჯერ განვიხილოთ თუ რა არის პრეკონდიცირება. მარტივ ენაზე რომ ვთქვათ, პრეკონდიცირება არის ტრანსფორმაცია, გადაყვანა ერთი ამოცანის სახისა სხვა ისეთში, რომლის ამოხსნაც შედარებით მარტივად, წრფივ დროში ხდება. ამ გადამყვან აპლიკაციას კი პრეკონდიციონერი ჰქვია (preconditioner [9]).

წრფივ სისტემებში, A მატრიცის პრეკონდიციონერი **P** - არის ისეთი მატრიცი, რომლისთვისაც $P^{-1}A$-ს აქვს ნაკლები კონდიციის რიცხვი (ფუნქციის მგრძნობელობა - რამდენად საგრძნობლად იცვლება შედეგი პატამეტრების მცირე ცვლილებით) ვიდრე A-ს. ზოგადად, პრეკონდიციონერს უწოდებენ არა P-ს არამდეს $T=P^{-1}$-ს, რადგანაც თავად P იშვიათად გვხვდება (გამოიყენება).

მაგალითად, Ax=b სისტემის ამოხსნის მაგივრად, მავანმა შეიძლება ამოხსნას მარჯვენა პრეკონდიციონირებული სისტემა:

$$A\,P^{-1}\,P\,x = b,$$

$A\,P^{-1}\,y = b$ -ის ამოხსნით y-ისთვის და

შემდგომ $P\,x = b$-ს ამოხსნით x-ისთვის.

ალტერნატიულად, შეუძლია ამოხსნას მარცხენა პრეკონდიცირებული სისტემა:

$$P^{-1}(Ax - b) = 0.$$

ორივე სისტემა იძლევა იგივე პასუხს, რასაც თავდაპირველი მანამ - სანამ პრეკონდიციონერი მატრიცი **P** არის შებრუნებადი (ანუ, $AP=PA=I_n$).

საზოგადოდ, მარცხენა პრეკონდიციონერი უფრო ხშირად გვხვდება ვიდრე მარჯვენა.

პრეკონდიცირებული სისტემის მიზანია კონდიციის რიცხვის შემცირება მარცხენა ან მარჯვენა პრეკონდიცირებული $P^{-1}A$ ან AP^{-1} მატრიცისთვის. იქიდან გამომდინარე, რომ ბევრნი აკეთებდნენ პრეკონდიციონერებს მეჩხერი ჩოლესკის გამოყენებით, ჩვენც გავარჩიეთ იგი.

ქოლესკის ფაქტორიზაცია და მეჩხერი ქოლესკი

ქოლესკის ფაქტორიზაცია (ანუ იგივე ქოლესკის დეკომპოზიცია) არის შემდეგი ოპერაცია:

$$\begin{aligned}\mathbf{A} = \mathbf{L}\mathbf{L}^T &= \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix} \\ &= \begin{pmatrix} L_{11}^2 & & (\text{symmetric}) \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix},\end{aligned}$$

აქედან, L-ის ელემენტებისთვის ვიღებთ შემდეგ ფორმულას:

$$\begin{aligned} L_{j,j} &= \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}, \\ L_{i,j} &= \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) \quad \text{for } i > j. \end{aligned}$$

ფესვქვეშა გამოსახულება ყოველთვის დადებითია თუ A მატრიცი არის დადებითად-განსაზღვრული და შედგება ნამდვილი რიცხვებისგან.

მაგალითად:

$$\begin{pmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 6 & 1 & 0 \\ -8 & 5 & 3 \end{pmatrix} \begin{pmatrix} 2 & 6 & -8 \\ 0 & 1 & 5 \\ 0 & 0 & 3 \end{pmatrix}.$$

ზოგადად, ქოლესკის დეკომპოზიცია გამოიყენება Ax=b სისტემის ამოსახსნელად. თუ A არის სიმეტრიული და დადებითად-განსაზღვრული, მაშინ ჩვენ შეგვიძლია ამოვხსნათ

Ax=b, თავდაპირველად, ქოლესკის დეკომპოზიციის გამოთვლით: A = LL*, შემდგომ Ly=b-ს ამოხსნით და ბოლოს, L*x=y-ის ამოხსნით.

მეჩხერი ქოლესკი (იგივე არასრული ჩოლესკის ფაქტორიზაცია) - არის ქოლესკის ფაქტორიზაციის მეჩხერი მიახლოება.

არასრული ქოლესკის ფაქტორიზაცია ხშირად გამოიყენება, როგორც პრეკონდიციონერი ისეთი ალგორითმებისთვის, როგორიცაა შეუღლებულ გრადიენტთა მეთოდი.

ალგორითმი მსგავსია უბრალო ჩოლესკის დეკომპოზიციისა, იმ განსხვავებით, რომ L-ში ჩანაწერი არანულოვანია, თუ შესაბამისი პოზიციის ელემენტი საწყის მატრიცშიც არანულოვანია. ამით, სიმეჩხერე მიღბული მატრიცისა მსგავსია საწყისის.

ალგორითმს აქვს შემდეგი სახე:

```
function a = ichol(a)
        n = size(a,1);

        for k=1:n
                a(k,k) = sqrt(a(k,k));
                for i=(k+1):n
                    if (a(i,k)!=0)
                        a(i,k) = a(i,k)/a(k,k);
                    endif
                endfor
                for j=(k+1):n
                    for i=j:n
                        if (a(i,j)!=0)
                            a(i,j) = a(i,j)-a(i,k)*a(j,k);
                        endif
                    endfor
                endfor
        endfor

    for i=1:n
        for j=i+1:n
            a(i,j) = 0;
        endfor
    endfor
endfunction
```

თავდაპირველად, ჩვენ გადავწერეთ არსებული ალგორითმი და იგი მოვარგეთ JNZ ფორმატს. ფორმატის სპეციფიკიდან გამომდინარე, საჭირო გახდა ციკლში დროებითი მასივის შემოღება სტრიქონისთვის, რამაც ალგორითმის მუშაობის დრო გაზარდა. ასევე, სტრიქონში ელემენტის სვეტის მოძებნა არსებულ ფორმატში არ ხდება მუდმივ დროში (მაგალითად, O(1)-ში) არამედ ყველაზე სწრაფად, თუ ელემენტები დალაგებულია, ხდება ლოგარითმულ დროში.

კოდის ფრაგმენტი (მეთოდი აბრუნებს მნიშვნელობებისა და ინდექსების მატრიცს):

```
std::pair<double**, int**>* CgSparse::cholesky() {
      // Construct L matrix's value and index arrays
```

```
int **L_Ind = (int**)malloc((n + 1) * sizeof(int*));
double** L_A = (double **)malloc(sizeof(double *) * n);
L_Ind[0] = (int*)malloc((n + 1) * sizeof(int));
L_Ind[0][0] = n;
for (int i = 0; i < n; i++) {
      // create temporary row
      double* tmpRow = (double*)calloc(n, sizeof(double));
      int nnzsInTmpRow = 0;
      for (int j = 0; j <= i; j++) {
        double sum = 0;
        if (j == i) { // summation for diagnols
            for (int k = 0; k < j; k++)
                  sum += pow(tmpRow[k], 2);
            // Find column index and element in A matrix
            int colIndex=iterativeSearch(Ind[j+1],0,Ind[0][j+1], j);
            tmpRow[j] =sqrt(((colIndex==-1)?0:A[j][colIndex])- sum);
            }
            else {
            ...
```

კვლევების მსვლელობისას ყურადღება მივაქციეთ ლინისა და მურის (Chin-Jen Lin, Jorge J. More) სტატიას[8] არასრული ქოლესკის ფაქტორიზაციის შესახებ შეზღუდული მეხსიერებით. სტატიაში განხილულია მეჩხერი ქოლესკის ალგორითმის გაუმჯობესებული ვარიანტი, რომელიც სტატიის ავტორების მტკიცებით ოპტიმალურია. ნაჩვენებია, რომ შეუღლებული გრადიენტის იტერაციები და გამოთვლითი დრო საგრძნობლად მცირდება p-ს პატარა მნიშვნელობებისთვის, სადაც p არის მატრიცის განზომილების ჯერადი რიცხვი.

ალგორითმს აქვს შემდეგი სახე:

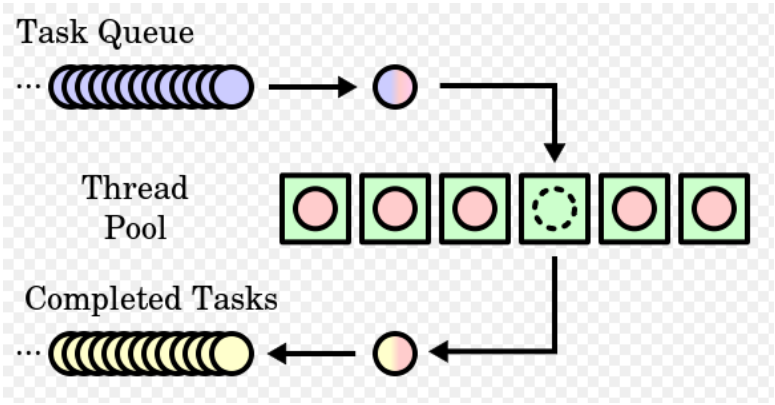
```
for j = 1:n
      a(j,j) = sqrt(a(j,j))
      col_len = size(i>j:a(i,j) 6= 0)
      for k = 1:j-1 & a(j,k) 6= 0
            for i = j+1:n & a(i,k) 6= 0
                  a(i,j) = a(i,j) - a(i,k)*a(j,k)
            end
      end
      for i = j+1:n & a(i,j) 6= 0
            a(i,j) = a(i,j)/a(j,j)
```

```
                a(i,i) = a(i,i) - a(i,j)^2
            end
        Retain the largest col_len + p elements in a(j+1:n,j).
End
```

გაპარალელება C++-ის ნაკადების გამოყენებით

თანამედროვე C++11-ის ვერსიას აქვს ჩაშენებული ნაკადებთან სამუშაო ბიბლიოთეკა <thread>(როგორც აქვთ გაჩუმებით სხვა პროგრამირების ენებს). რათქმაუნდა, აქამდეც არსებობდნენ ნაკადებთან სამუშაო სხვა ბიბლიოთეკები (მაგალითად, Boost-ის), მაგრამ ახლანდელი ენაში ჩაშენებული ბიბლიოთეკა მეტ სისწრაფეს გვპირდება.

ჩვენ პროექტში ვაპირებთ ნაკადების გამოყენებით გავაპარალელოთ შეუღლებულ გრადიენტთა მეთოდი და გამოვიყენოთ ნაკადებში ცნობილი ნაკადების დაგუბება (Thread Pool), რათა დავაჩქაროთ ნაკადების გამოყენებით ოპერაციების შესრულება. ნაკადის გამოყოფა მოიხმარს რესურსს. იმ შემთხვევაში, როდესაც ხდება პატარა ზომის პარალელური დავალებების შესრულება ნაკადების გამოყენებით, მიზანშეწონილია დაგუბებების ჩართვა. ამით თავიდან ვიცილებთ ნაკადების ბევრჯერ შექმნის აუცილებლობას მათი წინასწარი შექმნითა და ხელმეორედ გამოყენებით:



ნაკადების რაოდენობა განისაზღვრება პროგრამისთვის განკუთვნილი რესურსებიდან გამომდინარე: პარალელური პროცესორების რაოდენობის, ბირთვებისა და მეხსიერების მიხედვით.

GPU-ს პროცესორების მხარდაჭერა

შესაძლებელია კომპიუტერის არა მარტო ერთადერთი გამომთვლელი პროცესორის გამოყენება, არამედ გრაფიკული პროცესორებისაც. გრაფიკული დაფები აღჭურვილია მრავალი მიკრო პროცესორებით, რომლებიც ძირითადად გამოიყენება გრაფიკული და სივრცითი ელემენტების დასამუშავებლად და მათი დიდი ნაწილი დროის უმეტეს ნაწილში უმუშევარია (უსაქმოა). Nvidia-ს აქვს C++-ზე დაწერილი CUDA Toolkit [9], რომელიც საშუალებას გვაძლევს გრაფიკული დაფაზე არსებული პროცესორები გამოვიყენოთ, როგორც უბრალო პროცესორები.

CUDA გვხვდება სხვადასხვა ახალ აპლიკაციებში, მათ შორის ყველაზე თვალსაჩინოდ კრიპტო-ვალუტის მაინერებში. აქ, გრაფიკული დაფის პროცესორები გამოიყენება ჰეშის დასათვლელად (რომლიც მოძებნაც დიდ გამოთვლით რესურსს მოითხოვს).

სამომავლოდ ვაპირებთ CUDA-ს გამოყენებას ჩვენს პროექტში, რაც საინტერესო პროცესი იქნება და საინტერესო შედეგებსაც მოგვცემს.

გამოყენებული ლიტერატურა:

1. Jagged non-zero submatrix data structure. https://doi.org/10.1016/j.trmi.2017.10.002 (with Giga Chalauri, Vakhtang Laluashvili)
2. https://en.wikipedia.org/wiki/Sparse_matrix
3. Sparse Matrix File Formats - http://math.nist.gov/MatrixMarket/formats.html#hb
4. Sparse Matrix Collections - http://math.nist.gov/MatrixMarket/collections/hb.html
5. G. Gundersen, T. Steihaug, Data structures in Java for matrix computations, Concurrency and Computation: Practice and Experience, 2004, 799-815
6. Yousef Saad - Iterative Methods for Sparse Linear Systems (second ed.), SIAM (2003)
7. Preconditioner - https://en.wikipedia.org/wiki/Preconditioner
8. Incomplete Cholesky Factorization with Limited Memory \ http://ftp.mcs.anl.gov/pub/tech_reports/reports/P682.pdf
9. Nvidia CUDA - https://docs.nvidia.com/cuda/
10. https://pdfs.semanticscholar.org/cf79/2bb9ac27ea6b65aa31c091be1745555d0db9.pdf