ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო
უნივერსიტეტი

გიგა ჩალაური

მეჩხერი მატრიცების ფორმატების შედარებისა და დამუშავების გარემო

კომპიუტერული მეცნიერება

ნაშრომი შესრულებულია კომპიუტერული მეცნიერების მაგისტრის
აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: პროფესორი კობა გელაშვილი

თბილისი 2017

სარჩევი

სარჩევი ______ **2**

ანოტაცია ______ **3**

Abstract ______ **3**

1. *შესავალი* ______ **4**

2. *მეჩხერი მატრიცების გამოყენების პრაქტიკული მაგალითები და აპლიკაციები* ______ **8**

3. *მეჩხერი მატრიცების წარმოდგენის ფორმატები* ______ **10**

4. JNZ-ქვემატრიცის ტესტირების პარამეტრები* ______ **18**

5. *პროგრამული უზრუნველყოფის მოდულების აღწერა* ______ **20**

6. *რიცხვითი ექსპერიმენტები **jnz**-ქვემატრიცის ეფექტურობის გარკვევის მიზნით* ______ **23**

7. *რიცხვითი ექსპერიმენტები **cg**-მეთოდში **jnz**-ქვემატრიცის მეჩხერობის გარკვევის მიზნით* ______ **27**

8. *დასკვნები* ______ **28**

8.1. პატარა მატრიცები ______ **28**

8.2. საშუალო მატრიცები ______ **29**

8.3 დიდი მატრიცები ______ **30**

ანოტაცია

განსაზღვრულია ახალი მონაცემთა სტრუქტურა (ფორმატი) მეჩხერი მატრიცის შენახვისთვის - jnz-ფორმატი. ჩატარებულია ექსპერიმენტები სხვა ფორმატებთან შესადარებლად მისი ეფექტურობის გარკვევის მიზნით. ექსპერიმენტი მდგომარეობს წრფივ განტოლებათა სისტემის ამოხსნაში შეუღლებულ გრადიენტთა მეთოდით. შედეგებმა დაადასტურა, რომ როგორც შევსების, ასევე ამოხსნის დროის თვალსაზრისით ახალი ფორმატი უფრო ეფექტურია.

ექსპერიმენტების ჩატარება შესაძლებელი გახდა შესაბამისი გარემოს შექმნის შემდეგ, რაც აღწერილია წარმოდგენილ ნაშრომში და წარმოადგენს რთულ და ფაქიზ სისტემას.

ექსპერიმენტების კიდევ ერთი სერია არის ჩატარებული მატრიცის მეჩხერობის შინაარსის გარკვევის მიზნით.

Abstract

It is defined new data structure (format) for saving sparse matrixes – jnz-format. We conducted experiments for comparison with other formats in order to investigate effectiveness. The experiment is solving linear equation with conjugate gradient method. The results proved that new format is much better in term of filling time of matrix as well as solving.

Experimentation became possible after creating relevant environment, which is described in this work and represents complicated and delicate system.

There were conducted one more series of experiments in order to investigate the content of sparseness.

1. შესავალი

რიცხვით ანალიზში მეჩხერი ეწოდება მატრიცას, რომელშიც ელემენტების უმრავლესობა არის 0-ის ტოლი. არანულოვანი ელემენტების შეფარდებას მატრიცის ელემენტების საერთო რაოდენობასთან კი სიმეჩხრის ხარისხი ეწოდება. მეჩხერი მატრიცების თემა დღესდღეობით ძალიან აქტუალურია, რადგან რეალური ცხოვრების ბევრი ამოცანა ასეთი სახის მატრიცებზე ახორციელებს ოპერაციებს. მაგალითად: კომპიუტერული გრაფიკის ამოცანები, რეკომენდატორი სისტემები, სოციალური ქსელები, რუკებზე და გრაფებზე დაფუძნებული აპლიკაციები. თემის აქტუალურობას განსაზღვრავს მეჩხერი მატრიცების შენახვის ფორმატები. რადგან შეკრების არითმეტიკულ ოპერაციებში 0 არის ნეიტრალური ელემენტი, ლოგიკურია შეკითხვა: ხომ არ შეიძლება ასეთი ელემენტების ამოყრით მეხსიერების ეკონომიის გაკეთება ? ხოლო მეხსირების ეკონომია იქონიებს თუ არა გავლენას პროგრამების შესრულების სისწრაფეზე ?

მეჩხერი მატრიცები ძალიან ხშირად გვხვდება გამოყენებებში. გრაფებთან დაკავშირებულ ალგორითმებში ან კერძოწარმოებულიანი განტოლებების ამოხსნის პროცესში, მატრიცები ყოველთვის შეიცავს გარკვეული რაოდენობის ნულოვან ელემენტებს. მეჩხერი მატრიცების ინტენსიური შესწავლა მიმდინარეობს 1970-იანი წლებიდან და ამ დროის განმავლობაში დამუშავებულია რამდენიმე მონაცემთა სტრუქტურა (ფორმატი) მათი წარმოდგენისთვის. ზოგიერთი ფორმატი დამუშავებულია ისეთი შემთხვევებისთვის, როდესაც მეჩხერობა ვლინდება გარკვეული სისტემატიკური მოდელის (some systematic pattern) სახით (მაგალითად, 3 ან 5 დიაგონალური), ან როდესაც არანულოვანი ელემენტების განლაგება არ ექვემდებარება რაიმე კანონზომიერებას. წარმოდგენილ ნაშრომში განხილულია მეორე სახის მეჩხრობა. შევნიშნოთ, რომ ამ შემთხვევაში სიმეჩხერის გათვალისწინებით მეხსიერების ხარჯის და პროგრამის შესრულების დროის შემცირება შედარებით რთულია.

მეჩხერი მატრიცებისთვის დამუშავებული ფორმატებიდან რამდენიმე (ყველაზე საიმედო და სწრაფი) არის იმპლემენტირებული თანამედროვე დაპროგრამების ენების ბიბლიოთეკებში (იხ. მაგალითად [4]). ასეთი ბიბლიოთეკებიდან ერთ-ერთი ყველაზე საიმედო და ეფექტური არის boost, C++ ენის ცნობილი სამეცნიერო გარე ბიბლიოთეკა, რომლიდანაც პერიოდულად ხდება C++ ენის სტანდარტის გაფართოება. jnz-ქვემატრიცის ფორმატის ტესტირებისთვის ჩვენ გამოვიყენეთ boost ბიბლიოთეკის ორი ყველაზე სწრაფი Mapped Matrix და Compressed Matrix ფორმატი, რომლებსაც მოკლედ აღვწერთ შემდეგ პუნქტში.

jnz-ქვემატრიცა პრინციპულად განსხვავდება აღნიშნული ორი ფორმატისგან და აქტიურად იყენებს თანამედროვე პროგრამირების ენების შესაძლებლობებს, რაც უკავშირდება დინამიკურად შექმნილ ერთგანზომილებიან მასივებს და კბილებიან ორგანზომილებიან მასივებს. იდეურად იგი წარმოადგენს Ellpack-Itpack ფორმატის განზოგადებას (იხ. [5]) რომელიც ეფექტურად გამოიყენებოდა ისეთ კერძო შემთხვევებში, როდესაც წინასწარ იყო ცნობილი თითოეულ სტრიქონში არანულოვანი ელემენტების მაქსიმალური რაოდენობა. ამ შემთხვევებში საწყისი მატრიცის ნაცვლად შესაძლებელია განხილულ იქნას ორი შედარებით მცირე მართკუთხა მატრიცა, ერთი შედგენილი საწყისი მატრიცის არანულოვანი ელემენტებით, ხოლო მეორე - პირველი მატრიცის ელემენტების სვეტების ინდექსებს (საწყის მატრიცაში).

jnz-ქვემატრიცა ძალიან ახლოსაა Ellpack-Itpack ფორმატის სხვა განზოგადებასთან (იხ. [6]), რომელიც „ჯავას მეჩხერი მატრიცა"-ს სახელითაა ცნობილი და წარმოადგენს ერთგანზომილებიანი მასივებისგან შედგენილ ორ მასივს, რომლებიც მიიღება Ellpack-Itpack ფორმატიდან ნულების და მათი ინდექსების ამოყრით. თანამედროვე პროგრამირების ენებში ასეთი ორგანზომილებიანი მასივები კბილებიანი მატრიცის სახელითაა ცნობილი. [6]-ში და იგივე ავტორების სხვა ნაშრომებში ჩატარებულია გარკვეული ტესტები, რომლებიც ამტკიცებენ „ჯავას მეჩხერი მატრიცა"-ს

ეფექტურობას. თუმცა, უნდა აღინიშნოს, რომ ისინი მეხსიერების საჭირო მოცულობის გამოთვლის დროს არ ითვალისწინებენ გარკვეულ გარემოებებს.

მეორე სექცია ეთმობა მეჩხერი მატრიცების გამოყენების პრაქიკულ მაგალითებსა და აპლიკაციებს. მოკლედ იქნება განხილული რამდენიმე რეალური აპლიკაცია, რომელშიც ასეთი მატრიცები აქტიურად გამოიყენება.

მესამე სექცია ეთმობა jnz-ქვემატრიცის ფორმატს. მოკლედაა აღწერილი ჩვენვის საინტერესო ფორმატები. jnz-ქვემატრიცის გამოყენების შემთხვევაში ჩვეული მატრიცული ალგორითმების კოდები უმნიშვნელოდ იცვლება. იგი ძალიან ბუნებრივად ერგება მეჩხერი მატრიცებისთვის ტიპურ ოპერაციებს, მაგალითად სტრიქონების გადანაცვლებას, რაც ძალიან გავრცელებული მოქმედებაა ალგებრული ალგორითმების პარალელურ იმპლემენტაციებში.

რეალურ გამოყენებებში jnz-ქვემატრიცის ეფექტურობის გარკვევის მიზნით ჩვენს მიერ შერჩეულია შეუღლებულ გრადიენტების (მოკლედ cg) მეთოდი, რომელიც ძალიან ეფექტურია $Ax=b$ სახის სისტემებისთვის, სადაც A არის სიმეტრიული, დადებითად განსაზღვრული და მეჩხერი მატრიცა. cg-მეთოდის შერჩევაში იმ გარემოებამაც ითამაშა გარკვეული როლი, რომ სიმეტრიული მეჩხერი მატრიცისთვის jnz-ქვემატრიცა შეგვიძლია უფრო ეკონომიური გავხადოთ და შევინახოთ მხოლოდ დიაგონალი და ზედა სამკუთხა მატრიცის არანულოვანი ელემენტები. თუმცა უნდა შევნიშნოთ, რომ ამ დროს cg-მეთოდი 15-20%-ით ნელია jnz-ქვემატრიცის ტიპურ გამოყენებასთან შედარებით. განსხვავების მიზეზი ახსნილია პირველი სექციის ბოლოში.

მეოთხე სექციაში ძალიან მოკლედაა დახასიათებული ჩვენს მიერ გამოყენებული cg-მეთოდი. რადგან ჩვენთვის ალგორითმზე მეტად მის მიერ გამოყენებული მონაცემთა სტრუქტურაა საინტერესო, ამიტომ, არ განვიხილავთ პრეკონდიცირებულ ან დაპარალელებულ ვარიანტებს.

მეხუთე სექციაში მოკლედ იქნება აღწერილი პროგრამული უზრუნველყოფის მოდულების დანიშნულება. ის არის ერთი დიდი პროექტი, რომელიც არქიტექტურული თვალსაზრისის სისწორიდან გამომდინარე დაყოფილია ისეთ მოდულებად, რომლებსაც სხვადასხვა შინაარსის დავალება აქვთ გასაკეთებელი.

მეექვსე სექციაში, ვადარებთ cg-ალგორითმში მეჩხერი მატრიცის სამი სხვადასხვა ფორმატის გამოყენების შედეგებს: jnz-ქვემატრიცა, Mapped Matrix და Compressed Matrix. cg-ის სამივე ვარიანტი ეშვება [2]-იდან აღებულ 89 სხვადასხვა ზომის მეჩხერ მატრიცასა და შემთხვევითად გენერირებულ მარჯვენა მხარეებზე. Mapped Matrix და Compressed Matrix ფორმატები ჩვენს პროექტში ჩართულია boost ბიბლიოთეკიდან, რაც საშუალებას გვაძლევს გამოვიყენოთ იქ იმპემენტირებული მატრიცული ოპერაციების შესაბამისი ალგორითმები.

ტესტების შედეგების თვალსაჩინოებისთვის გამოყენებულია (იხ. [1]). შედეგები გვიჩვენებს რომ jnz-ქვემატრიცა ბევრად უკეთეს შედეგს იძლევა დანარჩენ ორ ფორმატთან შედარებით.

წარმოდგენილი ნაშრომის ბოლო ნაწილი ეთმობა cg-ალგორითმისთვის მატრიცაში ნულების პროცენტული რაოდენობის გარკვევას, რის შემდეგაც მატრიცა შეიძლება ჩაითვალოს მეჩხერად. აღმოჩნდა, რომ ნულების 20% და მეტი საკმარისია, რომ jnz-ქვემატრიცის გამოყენებამ მოგვცეს როგორც მეხსიერების, ასევე შესრულების დროის ეკონომია მატრიცის მკვრივი ფორმატის გამოყენებასთან შედარებით.

2. მეჩხერი მატრიცების გამოყენების პრაქტიკული მაგალითები და აპლიკაციები

მეჩხერი მატრიცები არის გული წრფივი ალგებრული სისტემების. ყველაფერი მნიშვნელოვანი რაც ხდება საკმარისად რთულ კომპიუტერულ სისტემებში მოითხოვს უამრავ წრფივ ალგებრულ ოპერაციებს. ჩვენ რეალურ შემთხვევებში არ შეგვიძლია შევინახოთ ძალიან დიდი განზომილების მქონე მატრიცები (რომელთა ელემენტების უმრავლესობა არის 0) და ჩავატაროთ მათზე ოპერაციები.

სფეროები, რომელშიც წრფივ ალგებრას წამყვანი როლი უკავია:

- კომპიუტერული გრაფიკა: ეს არის მარტივი „თამაში“, რომელშიც მატრიცების ერთმანეთზე/თავისთავზე გადამრავლებაა საჭრო.
- რეკომენდატორი სისტემები: ყოველთვის როცა Google ან Amazon გვთავაზობს რაღაცას, დარწმუნებული იყავით, რომ საჭიროა უამრავი მეჩხერი მატრიცის/ვექტორის გადამრავლება, რომ ის შედეგი მიიღოს რასაც გვთავაზობს.
- Machine Learning: თითქმის ყველა დასწავლადი მეთოდი ეყრდნობა მეჩხერი ვექტორების და მატრიცების ოპერაციებს.
- ინფორმაციის ძიება: ინდექსი არაფერია თუ არა ჭკვიანურად გაკეთებული მეჩხერი მატრიცა

ასევე არის უამრავი სხვა გამოყენება (სოციალური ქსელები, რუკებზე და გრაფებზე დაფუძნებული აპლიკაციები), სადაც გვჭირდება შევინახოთ კავშირი n ობიექტს შორის, ყველა მათგანი იყენებს მეჩხერ მატრიცებს.

ახლა განვიხილოთ ერთ-ერთი ყველაზე ცნობილი Google Matrix, რომელიც თავისი ბუნებიდან გამომდინარე არის მეჩხერი და გუგლი ამ მატრიცას იყენებს PageRank-ის ალგორითმში. Google Matrix არის სტოქასტური მატრიცის კერძო შემთხვევა. მატრიცა წარმოადგენს გრაფს, რომლის წიბოებიც წარმოადგენენ კავშირებს წიბოებს შორის.

თითოეული გვერდის რანკი/რეიტინგი შეიძლება დაგენერირდეს იტერაციულად Google Matrix-დან ახარისხების მეთოდით.

რომ დავაგენერიროთ Google Matrix G, პირველ რიგში უნდა დავაგენერიროთ მოსაზღვრეობის მატრიცა A, რომელიც წარმოადგენს კავშირებს გვერდებს ან წვეროებს (nodes) შორის. ჩავთვალოთ, რომ გვაქვს N ცალი გვერდი, ჩვენ შეგვიძლია შევავსოთ A შემდეგი წესის მიხედვით:

- A მატრიცის ელემენტი A[i][j] იქნება 1-ის ტოლი თუ კვანძ j-ს აქვს კავშირი/ლინკი კვანძ i-თან, წინააღმდეგ შემთხვევაში იქნება 0-ის ტოლი. ეს არის კავშირების მსოაზღვრეობის მატრიცა.
- დაკავშირებული მატრიცა S წარმოადგენს გადასვლებს მოცემული ქსელის მარკოვის ჯაჭვში. S იგება A-გან , j სვეტის ელემენტების გაყოფით k[j]-ზე, სადაც k[j] არის j წვეროდან გამავალი კავშრების/ლინკების ჯამური რაოდენობა. სვეტები, რომლებსაც აქვს 0-ები, ანუ არ აქვს გამავალი კავშირები, იცვლება მუდმივი მნიშვნელობით 1/N.

საბოლოოდ Google Matrix G შეიძლება გამოვსახოთ S-ის გამოყენებით

$$G_{ij} = \alpha S_{ij} + (1-\alpha)\frac{1}{N} \qquad (1)$$

სადაც α არის ჩახშობის კოეფიციენტი. როგორც წესი S არის მეჩხერი მატრიცა და თანამედროვე მიმართული ქსელისთვის მას აქვს დაახლოებით 10 არანულოვანი ელემენტი თითოეულ სვეტში ან სტრიქონში, აქედან გამომდინარე დაახლოებით 10N გადამრავლების ოპერაციაა საჭირო G მატრიცის ვექტორზე გადამრავლებისას.

3. მეჩხერი მატრიცების წარმოდგენის ფორმატები

მეჩხერი მატრიცების წარმოდგენის ფორმატებს შორის ერთ-ერთი ყველაზე ეფექტური არის CRF (შეკუმშული სტრიქონის ფორმატი). განვიხილოთ იგი [5]-ში მოტანილი მაგალითის გამოყენებით. CRF შემუშავებულია იმ დროს, როდესაც პროგრამირების ენები არ იყენებდნენ პოინტერებს. ეს ფორმატი არანულოვანი ელემენტების წარმოდგენისთვის იყენებს მასივს ელემენტების რაოდენობით nnz (გავრცელებული აღნიშვნა არანულოვანი ელემენტების რაოდენობისთვის). თუმცა, საწყის მართკუთხა მატრიცაში არანულოვანი ელემენტების პოზიციის აღდგენისთვის CRF ინახავს კიდევ ორ მთელრიცხვა მასივს, ელემენტების რაოდენობით nnz და n (სტრიქონების რაოდენობა).

მაგალითად, მარცხნივ წარმოდგენილი მატრიცის შესახებ მთელი ინფორმაცია შენარჩუნებულია მარჯვნივ წარმოდგენილ სამ მასივში:

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

```
AA ={1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.}.
JA ={0, 3, 0, 1, 3, 0, 2, 3, 4, 2, 3, 4},
IA ={0, 2, 5, 9, 11, 12}.
```

პირველ ორ მასივში ელემენტების რაოდენობა არის nnz. მასივი AA შედგება A მატრიცის არანულოვანი ელემენტებისგან, JA მასივის ელემენტები წარმოადგენენ AA მასივის შესაბამისი ელემენტების სვეტის ინდექსებს A მატრიცაში. IA მასივი რეკურსიულად განიმარტება: $\mathrm{IA}[0] = 0$ და $\mathrm{IA}[i] = \mathrm{IA}[i-1] + ($ $(i-1)$-ურ სტრიქონში არანულოვანი ელემენტების რაოდენობა). მასივებში ელემენტების მთლიანი რაოდენობა არის $2*nnz+n+1$. მეხსიერების მოცულობის გამოთვლისთვის არსებითია, რომ ბოლო ორი მასივი მთელრიცხვაა.

CRF ფორმატი საკმაოდ სწრაფია, რადგან იყენებს ერთგანზომილებიანი მასივებს. ამავე დროს, მისი გამოყენება პრობლემურია ძალიან დიდი ზომის მეჩხერი მატრიცებისთვის, რადგან ნებისმიერი კონფიგურაციის კომპიუტერისთვის ადვილია

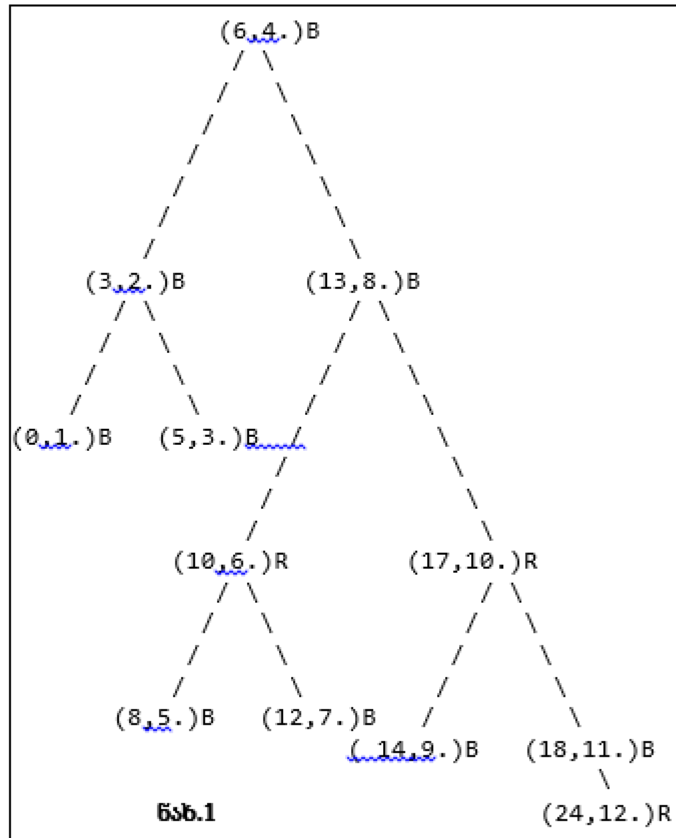
ისეთი მეჩხერი მატრიცის გენერირება, რომლის შესაბამისი არანულოვანი ელემენტებისთვის nz ელემენტებისთვის) ვერ მოხერხდება საჭირო მოცულობის მასივის გამოყოფა. გარდა ამისა, CRF ფორმატის შემთხვევაში მატრიცული ოპერაციების შესაბამისი კოდი საკმაოდ განსხვავდება იმისგან, რაც გვაქვს მკვრივი, მართკუთხა ფორმატის გამოყენების შემთხვევაში.

სავარაუდოდ ასეთი შემთხვევებისთვის, კარგად ცნობილი boost ბიბლიოთეკა CRF-ის გარდა (რომელიც აქ არის სახელით Compressed Matrix) კიდევ რამდენიმე ფორმატს გვთავაზობს, რომლებიც ელემენტების შესანახად იყენებენ წითელ-შავი ხის საფუძველზე შექმნილ ობიექტებს - ე. წ. mapped matrix. ასეთი ფორმატები შედარებით ნელა ივსება ელემენტებით და შედარებით ნელა ანხორციელებენ მატრიცულ ოპერაციებს (ყოველ შემთხვევაში იმ ოპერაციებს, რომლებიც გამოიყენება ჩვენს მიერ განხორციელებულ ტესტებში), სამაგიეროდ, ისინი ძალიან საიმედოდ მუშაობენ დიდი ზომის ამოცანებზე. boost ბიბლიოთეკის ყველაზე სწრაფი mapped matrix და CRF ფორმატებს ჩვენ ვიყენებთ jnz-ქვემატრიცის ფორმატის ეფექტურობის შეფასებისთვის.

იგივე A მატრიცის მაგალითზე, მისი შესაბამისი mapped matrix წარმოადგენს წითელ-შავ ხეს, რომელიც აგებულია შემეგი წყვილებისგან:

$$\{(0,1.),(3,2.),(5,3.),(6,4),(8,5.),(10,6.),(12,7.),(13,8.),(14,9.)(17,10.),(18,11.),(24,12.)\}$$

რადგან ყოველი არანულოვანი A[i][j] ელემენტის შესაბამისი გასაღები წითელ-შავ ხეში გამოითვლება ფორმულით $n*i+j$. შესაბამისად, წითელ-შავი ხის აგების სტანდარტული ალგორითმის მიხედვით, ამ მონაცემებით აგებულ ხეს აქვს ნახაზ 1-ზე წარმოდგენილი სახე. ხის კვანძებში ჩაწერილ წყვილებთან ერთად მითითებულია კვანძის ფერიც.



ნახ.1

jnz-ქვემატრიცა არსებითად განსხვავდება აღნიშნული ორი ფორმატისგან და აქტიურად იყენებს თანამედროვე პროგრამირების ენების შესაძლებლობებს დინამიკურ ერთ და ორ განზომილებიან მასივებთან დაკავშირებით. იდეურად, იგი წარმოადგენს Ellpack-Itpack ფორმატის განზოგადებას, რომელიც გამოიყენებოდა იმ შემთხვევაში, როდესაც წინასწარ იყო ცნობილი თითოეულ სტრიქონში არანულოვანი ელემენტების მაქსიმალური რაოდენობა Nd. ამ შემთხვევაში საწყისი მატრიცის ნაცვლად შესაძლებელია ორი მართკუთხა მატრიცის განხილვა, თითოეული ზომებით $n \times Nd$. პირველ მატრიცაში არის საწყისი მატრიცის არანულოვანი ელემენტები, აუცილებლობის შემთხვევაში ბოლოდან შევსებული ნულებით Nd რაოდენობამდე. მეორე მატრიცა შედგება მთელებისგან და შეიცავს პირველი მატრიცის ელემენტების სვეტების ინდექსებს (საწყის მატრიცაში). მაგალითად, შემდეგი მატრიცა, რომელსაც აქვს სამი დიაგონალი,

$$A = \begin{pmatrix} 1. & 0. & 2. & 0. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 0. & 6. & 7. & 0. & 8. \\ 0. & 0. & 9. & 10. & 0. \\ 0. & 0. & 0. & 11. & 12. \end{pmatrix}$$

შესაძლებელია წარმოვადგინოთ ორი მატრიცით:

$$COEF = \begin{pmatrix} 1. & 2. & 0. \\ 3. & 4. & 5. \\ 6. & 7. & 8. \\ 9. & 10. & 0. \\ 11. & 12. & 0. \end{pmatrix} \qquad JCOEF = \begin{pmatrix} 0 & 2 & 0 \\ 0 & 1 & 3 \\ 1 & 2 & 4 \\ 2 & 3 & 3 \\ 3 & 4 & 4 \end{pmatrix}$$

მეორე მატრიცა შედგება არანულოვანი ელემენტის სვეტის ინდექსისგან. აქვე, პირველის არანულოვან ელემენტებს შორის დამატებული ნულების შესაბამის პოზიციაში შეგვიძლია ნებისმიერი მთელი რიცხვი ჩავსვათ, კერძოდ, სტრიქონის ნომერი.

[5]-ში, დაპროგრამების ენა Java-ს საშუალებების საფუძველზე Ellpack-Itpack ფორმატი ეფექტურადაა განზოგადებული. Java-ში, ისევე როგორც C ენის საფუძველზე შექმნილ სხვა ენებში, ორგანზომილებიანი მასივი წარმოადგენს ერთგანზომილებიან მასივს, რომლის ელემენტები ასევე ერთგანზომილებიანი მასივებია. ჯავაში 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 სვეტის ინდექსები. [5]-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, საჭირო მასივების სტრიქონების რაოდენობის შენახვა.

მაგალითად, იგივე მატრიცა, რაც ჩვენ განვიხილეთ Ellpack-Itpack ფორმატის მაგალითად, 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},{1,2,4},{2,3},{3,4}};
```

ვიზუალურად ეს ფორმატი შეგვიძლია წარმოვიდგინოთ კბილებიანი მატრიცის სახით:

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

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

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

C++ ენაში შესაძლებელია „ჯავას მეჩხერი მასივი“-ს იდენტური ფორმატის მოწყობა, მაგრამ ეს არ იქნება საუკეთესო არჩევანი. ჩვენ ვქმნით ოდნავ განსხვავებული ფორმის კბილებიანი მატრიცების წყვილს, რომელსაც ვუწოდებთ jnz-ქვემატრიცას. 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 (ეს ყველაფერი შეგიძლიათ იხილოთ github-ზე, პროექტის ფაილებში).

jnz-ქვემატრიცის ფორმატის გამოყენების შემთხვევაში შესანახი არის $2*nnz+3*n+4$ ელემენტი, აქედან მხოლოდ nnz რაოდენობა არის ნამდვილი რიცხვი. $2*n+1$ არის პოინტერი, 2 ორმაგი პოინტერი, დანარჩენი - მთელი რიცხვები. მთელ რიცხვებს შორის, მხოლოდ ერთი იღებს სავარაუდოდ დიდ მნიშვნელობას. დანარჩენები აღწერენ სტრიქონებში ელემენტების რაოდენობას და არანულოვანი ელემენტების სვეტის ნომრებს. სავარაუდოდ, C ენაში მათთვის შესაძლებელია შედარებით ეკონომიური მოკლე მთელი რიცხვის ტიპის გამოყენება.

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. ეს მეთოდი ძალიან გავრცელებულია კომპიუტერული ალგებრის სისტემებში BLAS და LAPACK (იხ. [7], მაგალითად). მეჩხერი მატრიცის და ვექტორის ნამრავლი ჩვენი ტესტებისთვის არსებითია, რადგან cg-მეთოდი ძირითადად ამ ქვეპროგრამას ატრიალებს. აქ ჩვენ მოვიყვანთ jnz-ქვემატრიცის ფორმატში წარმოდგენილი მეჩხერი მატრიცის ვექტორზე გამრავლების ფუნქციის კოდს, 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 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.

ამ ფუნქციას ჩვენ ვიყენებთ ტესტების მეორე ჯფუფში, სადაც ვარკვევთ თუ თუ cg-ალგორითმისთვის ნულების რა პროცენტული შემცველობიდან იწყება მეჩხერი მატრიცები.

ტესტების პირველ ჯგუფში, სადაც ჩვენ მეჩხერი მატრიცების ფორმატების ეფექტიანობას ვადარებთ cg-ალგორითმისთვის, ვიყენებთ jnz-ქვემატრიცის 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 10-20%-ით ნელი ხდება.

4. JNZ-ქვემატრიცის ტესტირების პარამეტრები

რეალურ გამოყენებებში jnz-ქვემატრიცის ეფექტურობის მიზნით ჩვენს მიერ შერჩეულია შეუღლებულ გრადიენტთა (მოკლედ cg) მეთოდი. იგი ძალიან ეფექტურია $Ax = b$ სახის სისტემებისთვის, სადაც A არის სიმეტრიული, დადებითად განსაზღვრული და მეჩხერი მატრიცა. მეთოდის ალგორითმი საკმაოდ მარტივია, ამიტომ იმპლემენტაციაში მატრიცის ერთი წარმოდგენის ჩანაცვლება მეორით სირთულეს არ წარმოადგენს. cg-მეთოდის შერჩევაში იმ გარემოებამაც ითამაშა გარკვეული როლი, რომ სიმეტრიული მეჩხერი მატრიცისთვის jnz-ქვემატრიცა შეგვიძლია უფრო ეკონომიური გავხადოთ და შევინახოთ მხოლოდ დიაგომალი და ზედა სამკუთხა მატრიცის არანულოვანი ელემენტები. თუმცა უნდა შევნიშნოთ, რომ ამ დროს cg-მეთოდი 10-20%-ით ნელია ჩვეულებრივი წესით შენახულ jnz-ქვემატრიცის გამოყენებასთან შედარებით. განსხვავების მიზეზი ახსნილია ქვემოთ.

ჩვენს შემთხვევაში, cg-მეთოდი წარმოადგენს მხოლოდ მეჩხერი მატრიცების სხვადასხვა ფორმატის შედარების საშუალებას, ამიტომ არ ვცდილობთ მისი ყველაზე დახვეწილი და სწრაფი ვარიანტების დაპროგრამებას (პრეკონდიცირების და დაპარალელების გათვალისწინებით). პირიქით, ჩვენთვის მისაღებია მაქსიმალურად მარტივი კოდი, რომ ყურადღება გამახვილდეს მონაცემთა სტრუქტურებზე. cg-მეთოდის დაპროგრამება ადვილია, ხოლო ჩვენი ექსპერიმენტების კოდები მოთავსებულია github-ზე ამიტომ ამ საკითხზე აღარ გავამახვილებთ ყურადღებას.

მატრიცები აღებულიაა ფლორიდის კოლექციიდან, ხოლო $Ax = b$ გამტოლებათა სისტემის მარჯვენა მხარეების აღებულია შემთხვევითად, თუმცა მარჯვენა მხარის დიაპაზონი შეთანხმებულია მატრიცის დიაპაზონთან (შუალედი მინიმალურ და მაქსიმალურ ელემენტებს შორის). ჯამში გამოყენებულია 88 მატრიცა, რომლებიც პირობითად დაყოფილია სამ ჯგუფად:

- პატარები, რომლებშიც არანულოვანი ელემენტების რაოდენობა ნაკლებია 2000-ზე. ასეთი სახის 32 მატრიცაა.

- საშუალოები, რომლებშიც არანულოვანი ელემეტების რაოდენობა მეტია ან ტოლი 2000-ზე და ნაკლებია 5000-ზე. ასეთი სულ 32 მატრიცაა.
- დიდები, რომლებშიც არანულოვანი ელემენტების რაოდენობა მეტია ან ტოლი 5000-ზე. ასეთი სახის 24 მატრიცაა.

5. პროგრამული უზრუნველყოფის მოდულების აღწერა

როგორც ზემოთ თავებში იყო ნახსენები, პროგრამული უზრუნველყოფა წარმოადგენს მოდულარულ სისტემას, რომელიც ეფუძნება დიდი პროექტების აგების თანამედროვე მეთოდებსა და პრინციპებს. სისტემა დაყოფილია სამ ლოგიკურად მსგავს მოდულებად, რომელთაგან თითოეული ასრულებს კონკრეტულ დავალებას. პროექტის ასეთი დანაწევრება ბევრ უპირატესობას გვაძლევს პროექტის მართვის (ცვლილებების შეტანის, ახალი კოდის დამატების, რედაქტირების) დროს.

SparseLib მოდული მთლიანობაში წარმოადგენს ბიბლიოთეკას, რომელიც შეიცავს პროგრამული უზრუნველყოფისთვის საჭირო კლასებსა და მეთოდებს. ძირითადად, იგი შედგება ე.წ. სოლვერებისაგან, ანუ ამომხსნელი კლასებისაგან, რომლებსაც მთვარი აპლიკაცია იყენებს. უნდა აღვნიშნოთ, რომ ობიექტზე ორიენტირებული პროგრამირების პრინციპებიდან გამომდინარე, თითოეული სოლვერი დაყოფილია ორ ფაილად: Header და CPP ფაილებად. ჩვენთვის უცხო არ არის, რომ Header ფაილში არის კლასის აღწერა, ხოლო CPP ფაილში შესაბამისი მეთოდების იმპლემენტაცია. იმისათვის რომ კოდი უფრო მოხერხებული და დაცული გაგვეხადა ჩვენ შემოვიღეთ Evaluator (აბსტრაქტული) კლასი, რომელიც, როგორც უკვე ხვდებით, პოლიმორფიზმის დროს გამოიყენება. ეს კლასი აღწერს სავალდებულო მეთოდების პროტოტიპებს, რომლებსაც შვილი კლასები უნდა განსაზღვრავდნენ. სავალდებულო მეთოდები აბრუნებენ შესრულების დროებს. დროის მთვლელი ცალკე TimeCounter კლასში გვაქვს გატანილი. რაც შეეხება Helper კლასს, ის გვეხმარება მატრიცის შევსებაში. პრიმიტიული შევსების ოპერაცია დიდი მოცულობის მონაცემებისთვის მოუხერხებელია, რადგანაც დიდ დროს ანდომებს. ამ მიზეზით და პროექტის განვითარებასთან ერთად ვცდილობდით, დიდი მოცულობის ფაილებიდან წაკითხვის აჩქარებას. ჩვენმა ჩატარებულმა ექსპერიმენტებმა აჩვენა, რომ მეთოდი, რომელიც გამოყენებულია Helper.CPP ფაილში არის ერთ-ერთი საუკეთესო.

UnitTests მოდული ადვილი მისახვედრია, რომ წარმოადგენს ავტომატური ტესტირების მოდულს. სხვადასხვა პროგრამირების ენისთვის არსებობს

ერთეულოვანი ტესტების შესაქმნელი უამრავი ბიბლიოთეკა, რაც C++-ისთვისაც ზევრია (მაგალითად, boost-ის ტესტების ბიბლიოთეკა). ჩვენ ავირჩიეთ NativeTests, რომელიც Visual Studio-ს მოყვება. დღესდღეისობით რთული წარმოსადგენია დიდი სისტემები ტესტების მექანიზმის გარეშე, რადგან მანუალური გატესტვის პირობებში ადამიანური ფაქტორებიდან გამომდინარე შეცდომის დაშვების ალბათობა ძალიან დიდია. შესაბამისად, Test Driven Development-ზე დაფუძნებული სისტემის კორექტულობა უფრო სანდოა, ვიდრე ამის გარეშე. სწორედ ეს იყო მიზანი, შეგვექმნა შესაბამისი მოდული, რომელიც ჩვენს მიერ ხელით გენერირებულ მატრიცაზე და მარჯვენა მხარეზე, დაადგენდა თითოეული სოლვერისთვის, რომ ისინი მართლაც კორექტულად მუშაობენ. ანუ სხვა სიტყვებით, რომ ვთქვათ, ავიღეთ წინასწარ ამოხსნილი წრფივ განტოლებათა სისტემა და გავატარეთ თითოეულ ამომხსნელში, შემდეგ კი მიღებული ამონახსნები შევადარეთ რეალურებს. სწორედ ამ იდეას ეფუძნება Unit Test-ების იდეა.

Sparse Project მოდული წარმოადგენს ცენტრალურ მოდულს, რომელიც ერთმანეთთან აკავშირებს ტესტირების მოდულსა და ბიბლიოთეკების მოდულს. ასევე ამ მოდულის ერთ-ერთ საქაღალდეს წარმოადგენს მატრიცების კოლექცია, რომელიც ასევე ქვე-საქაღალდეებშია ჩალაგებული კატეგორიების მიხედვით. მოგეხსენებათ, რომ ყველა სისტემას, რომელიც რაიმე ამოცანას ხსნის გააჩნია შედეგების ნახვის შესაძლებლობაც. ჩვენ სისტემასაც გააჩნია ეს. შედეგები, რომლებსაც პროგრამა გვიბრუნებს (კერძოდ, ამოცანის ამოხსნის დროები კონკრეტული ზომის მასივებისთვის) ჩვენ შეგვეძლო მხოლოდ კონსოლის ინტერფეისში გვეჩვენებინა, მაგრამ ასეთი სახით ინფორმაციის აღქმა რთული და მოუხერხებელია. ჩვენ გადავწყვიტეთ ინფორმაცია შეგვენახა ისეთ უნივერსალურ ფორმატში, რომელსაც სხვადასხვა პროგრამირების ენა მარტივად დაამუშავებდა და წარმოგვიდგენდა. JSON იყო ერთ-ერთი ვარიანტი, რომელზეც ჩვენ შევჩერდით (შესაძლებელი იყო აგვერჩია XML ფორმატიც, თუმცა JSON უფრო მოხერხებული და ფართოდ გავრცელებულია). ჩვენი პროგრამა კონსოლში ინფორმაციის გამოტანასთან ერთად აგებს JSON

დოკუმენტს, რომელსაც ვინახავთ შესაბამის results.json ფაილში. ჩვენ შეგვიძლია ეს ინფორმაცია დავამუშავოთ და ვიზუალურად წარმოვადგინოთ ისე, რომ მარტივად აღქმადი და შედარებით დეტალური იყოს. ამისათვის, ჩვენ ვიყენებთ ვებ-ბრაუზერს, რომელიც ყველა ოპერაციულ სისტემაში არის ჩაშენებული. არსებულ მოდულში გვაქვს index.html ფაილი (შესაბამისი *.css ფაილებით), რომელიც გვიხატავს results.json ფაილში არსებულ ინფორმაციას. შედეგების ასე განცალკევება ცნობილი პრაქტიკაა. ასეთი მიდგომის გამოყენებით, სამომავლოდ ინფორმაციის გაზიარება სხვადასხვა სისტემებისთვის გაცილებით მარტივად მოხდება.

6. რიცხვითი ექსპერიმენტები jnz-ქვემატრიცის ეფექტურობის გარკვევის მიზნით

ძირითადი მეთოდი პროგრამული უზრუნველყოფის ოპტიმიზაციის მახასიათებლების შედარებაში განხილულია დოლანისა და მორეს სტატიაში, რომელსაც ჰქვია „benchmarking optimization software with performance profiles" (იხ. [1]). ამ მიდგომის გამოსაყენებლად საჭიროა ორი სიმრავლე: ერთი ტესტ პრობლემების სიმრავლე და მეორე - მეთოდების.

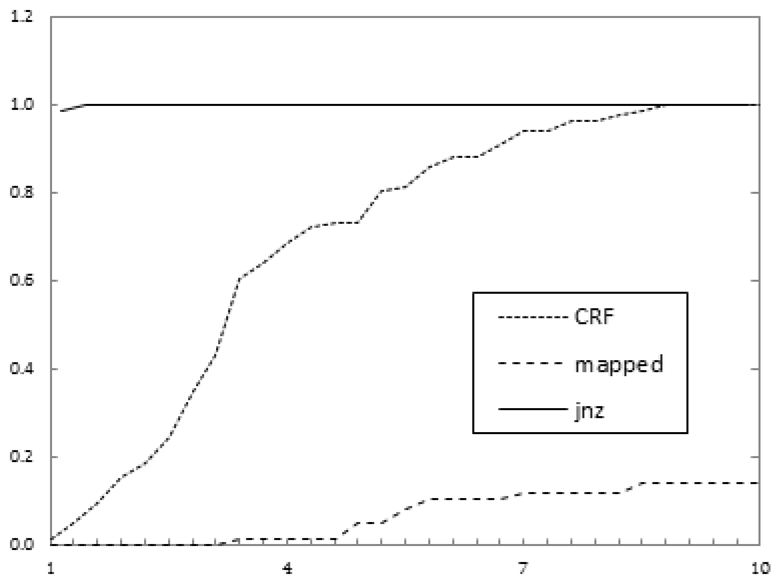
ყოველი სიმეტრიული, დადებითად განსაზღვრული მეჩხერი მატრიცა და შესაბამისი განზომილების მქონე ვექტორი ცალსახად განსაზღვრავს $Ax=b$ სახის სისტემას, რომელიც ამოხსნადია cg-მეთოდით. სატესტო ამოცანების სიმრავლე ჩვენს ექსპერიმენტებში შედგება დაახლოებით 90 ამოცანისგან - სიმეტრიული, დადებითად განსაზღვრული მეჩხერი მატრიცა და შესაბამისი განზომილების მქონე ვექტორი. 89 ასეთი მატრიცა აღებულია [8]-იდან. ისინი პირობითად შეგვიძლია გავყოთ სამ ჯგუფად: მცირე, საშუალო და დიდი ზომის. ყოველი მატრიცისთვის შემთხვევითად არის გენერირებული შესაბამისი განზომილების ვექტორი, რომელიც ყოველთის განიხილება ამ მატრიცასთან ერთად. შერჩეული მატრიცების ხილვა ასევე შესაძლებელია github-ზე.

$Ax=b$ 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 სოლვერი დაფიქსირებულია, ჩვენ პრაქტიკულად მხოლოდ მონაცემთა სტრუქტურების შედარებას ვახდენთ.

ექსპერიმენტების შედეგებში მონაწილეობს სამი ფორმატი. ერთი არის jnz-ქვემატრიცა, რომელიც ჩვენს მიერ არის იმპლემენტირებული რამდენიმე ალგებრულ ალგორითმთან ერთად (რაც საჭიროა cg-მეთოდში მისი ინტეგრაციისთვის), ორი სხვა

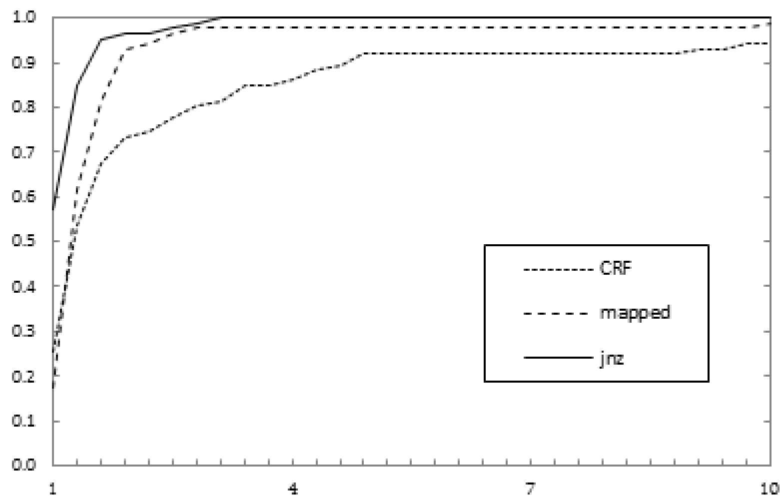
არის boost ბიბლიოთეკის Mapped Matrix და Compressed Matrix ფორმატები. სინამდვილეში ჩვენს მიერ ტესტირებული იყო ბევრად მეტი ფორმატი, მაგრამ სიმარტივისთვის დავტოვეთ მხოლოდ ყველაზე სწრაფები.

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

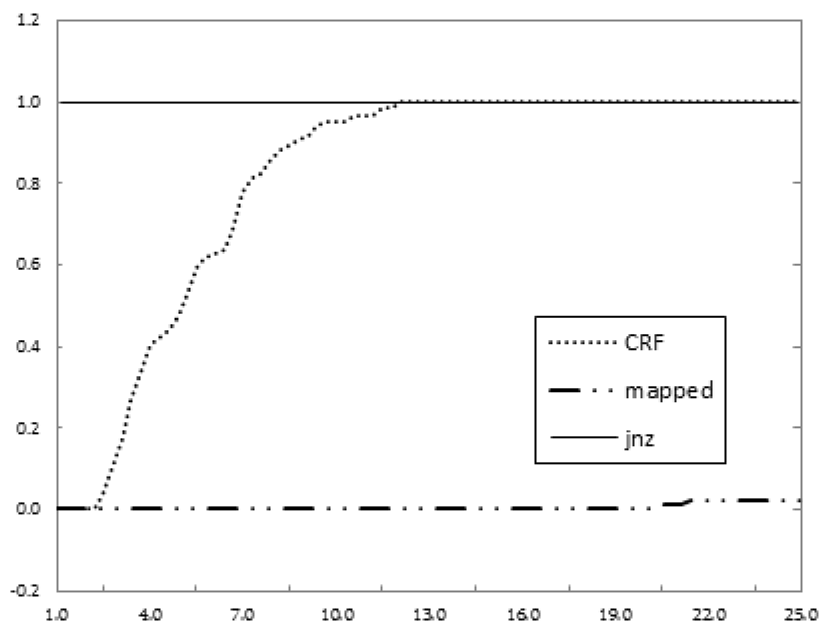
მეორე პროფაილი აგებულია განხილულ ამოცანებში ფაილიდან მატრიცების შევსების დროების მიხედვით. როგორც ცნობილია, დიდი ზომის მატრიცებისთვის ფაილიდან მონაცემების წაკითხვა შრომატევადი და ხანგრძლივი პროცედურაა, ამიტომ მისი განხორციელება არ არის ხელსაყრელი სტანდარტული მეთოდების გამოყენებით. ჩვენს მიერ გამოყენებული მეთოდის ხილვა აგრეთვე შესაძლებელია პროექტის კოდთან ერთად:



ფიგურა 2 Performance profile for filling times

როგორც ვხედავთ, შედარებით ახალი ფორმატები ბევრად სწრაფად ივსება CRF ფორმატთან შედარებით.

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



ფიგურა 3 Performance profile for solving times

იმის გამო, რომ ჩვენი ფორმატისთვის the distribution function for the performance ratio თითქმის ერთის ტოლია, ხოლო Mapped Matrix-ისთვის თითქმის ნულის ტოლი, ამიტომ ვერტიკალური ღერძი შედარებით ვრცელი მონაკვეთითაა წარმოდგენილი.

7. რიცხვითი ექსპერიმენტები cg-მეთოდში jnz-ქვემატრიცის მეჩხერობის გარკვევის მიზნით

ზოგიერთი განმარტების მიხედვით, მეჩხერი ეწოდება მართკუთხა მატრიცას, რომელშიც ელემენტებს შორის ნულების რაოდენობა აღემატება არანულოვანებისას. წინააღმდეგ შემთხვევაში მატრიცას ეწოდება მკვრივი (dence). ზოგიერთი განმარტების მიხედვით, კვადრატულ მატრიცას n სტრიქონით ეწოდება მეჩხერი, თუ მასში არანულოვანი ელემენტების რაოდენობა არის $O(n)$ რიგის. [5] -ში გამოთქმულია მოსაზრება: „მეჩხერი მატრიცა განისაზღვრება, გარკვეულწილად ბუნდოვნად, როგორც მატრიცა, რომელსაც აქვს ძალიან ცოტა არანულოცანი ელემენტი. მაგრამ, პრაქტიკაში, მატრიცას შეიძლება ეწოდოს მეჩხერი, როცა სპეციალური ტექნიკა შეიძლება გამოყენებული იქნას ნულოვანი ელემენტების დიდი რაოდენობის და მათი პოზიციიდან გარკვეული სარგებლის მიღების თვალსაზრისით“. სავარაუდოდ, ეს ნიშნავს რომ მეჩხერი მატრიცის „აბსოლუტური“ განმარტება არ არსებობს, მატრიცის მეჩხერობა არის ფარდობითი კატეგორია და დამოკიდებულია იმ ალგორითმზე, რომელიც ამ მონაცემთა სტრუქტურას იყენებს.

ჩვენი ექსპერიმენტების შედეგები აჩვენებს, რომ cg-ალგორითმისთვის მატრიცა ჩაითვლება მეჩხერად, თუ არანულოვანების რაოდენობა მეტია ან ტოლი 20%-ზე.

8. დასკვნები

ჩვენს მიერ ჩატარებული ცდებიდან კარგად გამოჩნდა, რომ nz-ქვემატრიცა იდეალურადაა მორგებული მატრიცულ ოპერაციებს. ქვემოთ მოყვანილ სურათებზე ნაჩვენებია ცხრილების სახით, თუ რომელმა სოლვერმა რომელი მატრიცისთვის რა დრო მოანდომა შევსებას და ამოხსნას, ცალ-ცალკე.

8.1. პატარა მატრიცები

#		N	NNZ	Fill	Solve	Fill	Solve	Fill	Solve
1	LFAT5.mtx	14	30	0.385337	0.0727754	0.315671	0.00870817	0.380982	0.00870817
2	LFAT5_E.mtx	14	30	0.580026	0.0478949	0.387514	0.00746415	0.512849	0.00746415
3	LFAT5_M.mtx	14	30	0.886989	0.0852157	0.727443	0.00964119	0.57194	0.00590912
4	LF10.mtx	18	50	0.620768	0.0982779	0.463399	0.0124402	0.600553	0.00870817
5	LF10_E.mtx	18	50	0.772228	0.0687323	0.627921	0.0118182	0.779692	0.00746415
6	LF10_M.mtx	18	50	0.875171	0.0640673	0.734596	0.0102632	0.865841	0.00715314
7	Trefethen_20.mtx	20	89	0.50694	0.0475839	0.352992	0.00964119	0.360456	0.0105742
8	ex5.mtx	27	153	1.83774	0.434165	1.57276	0.0687323	1.80446	0.0208374
9	bcsstk01.mtx	48	224	2.12417	0.60584	1.76278	0.182872	2.03336	0.273063
10	mesh1e1.mtx	48	177	3.38966	0.144307	1.85018	0.0149283	2.16989	0.0457179
11	mesh1em1.mtx	48	177	2.18264	0.142752	1.81068	0.0230145	2.03025	0.0905028
12	mesh1em6.mtx	48	177	2.02185	0.133422	1.64647	0.0152393	2.00257	0.051005
13	bcsstm02.mtx	66	66	1.32986	0.0677993	1.02632	0.0155503	1.32022	0.0566031
14	nos4.mtx	100	347	3.14956	0.724644	3.23073	0.109785	4.57179	0.195001
15	bcsstk03.mtx	112	376	3.44066	6.02637	4.09315	0.962564	3.3853	3.3937
16	bcsstk22.mtx	138	417	5.2672	5.45443	4.1709	0.646582	8.58595	4.44894
17	bcsstm22.mtx	138	138	2.512	0.520002	1.80321	0.0559811	2.66719	1.08945
18	bcsstm05.mtx	153	153	3.82786	0.360145	1.77553	0.0255025	2.58011	0.46993
19	nos1.mtx	237	627	2.76671	58.7515	2.46441	11.8192	4.36124	2.69642
20	mesh3e1.mtx	289	1089	3.88416	0.658711	1.98857	0.105431	3.36602	0.851224
21	mesh3em5.mtx	289	1089	7.3046	0.443495	5.02741	0.0877037	7.50085	1.0991
22	bcsstm06.mtx	420	420	10.4433	4.11772	5.84971	0.673328	9.92794	30.3747
23	bcsstm20.mtx	485	485	7.47472	11.5856	4.96832	1.11091	7.70766	139.639
24	494_bus.mtx	494	1080	12.416	76.2142	8.09735	12.704	13.3322	214.239
25	bcsstm19.mtx	817	817	13.4603	32.9928	8.26374	5.12351	12.4315	595.263
26	bcsstm08.mtx	1074	1074	13.1223	16.4342	4.85325	1.55596	10.2607	293.775
27	bcsstm09.mtx	1083	1083	24.3157	0.234499	14.5206	0.0864597	20.7295	3.51033
28	bcsstm11.mtx	1473	1473	23.2704	2.67092	17.2863	0.463399	23.2331	70.2267
29	bcsstm26.mtx	1922	1922	32.6127	383.121	30.8167	41.8549	36.6903	11465

8.2. საშუალო მატრიცები

#		N	NNZ	Fill	Solve	Fill	Solve	Fill	Solve
1	bcsstk02.mtx	66	2211	35.6369	1.35381	23.7276	0.575983	28.8915	0.219259
2	bcsstk04.mtx	132	1890	19.7439	11.6596	18.1814	3.60425	19.8403	3.69569
3	lund_a.mtx	147	1298	5.65907	5.63356	4.57179	1.43903	7.7736	1.17218
4	Trefethen_150.mtx	150	1095	3.02796	1.64118	1.84396	0.359523	2.61649	2.73872
5	bcsstk05.mtx	153	1288	13.6081	4.82059	11.9896	1.1784	13.5894	2.26692
6	Trefethen_200.mtx	200	1545	4.32765	2.64666	2.65288	0.591534	5.99682	6.63656
7	Trefethen_300.mtx	300	2489	7.23462	5.33873	4.12829	1.20235	5.57043	18.5472
8	mesh2e1.mtx	306	1162	12.3675	2.7776	10.5369	0.449404	20.6847	8.82418
9	mesh2em5.mtx	306	1162	15.0906	2.07254	10.6199	0.345528	12.4925	9.12647
10	plat362.mtx	362	3074	43.5362	540.957	45.2781	121.872	48.4939	6.60017
11	mhdb416.mtx	416	1364	14.7395	194.438	16.5194	29.6746	24.3452	770.544
12	bcsstk06.mtx	420	4140	48.8087	223.909	57.668	72.0772	44.3314	76.7199
13	bcsstk07.mtx	420	4140	41.0593	207.867	37.2996	56.6927	41.7921	77.4281
14	bcsstm07.mtx	420	3836	57.6129	18.154	44.4919	4.15535	62.9884	37.2906
15	nos5.mtx	468	2820	21.0744	33.1103	14.5884	5.66218	17.9388	12.3308
16	bcsstk20.mtx	485	1810	12.871	2974.62	11.7937	482.597	12.7366	774.177
17	494_bus.mtx	494	1080	10.675	62.7125	7.54221	7.80128	10.456	194.823
18	Trefethen_500.mtx	500	4489	14.0366	11.9398	7.35654	2.96451	11.3443	62.7971
19	662_bus.mtx	662	1568	14.8166	36.769	15.2838	5.24263	16.1649	135.932
20	nos6.mtx	675	1965	11.533	100.954	12.2533	12.3731	11.2976	88.7076
21	685_bus.mtx	685	1967	25.4167	46.5617	20.3659	8.18164	29.5639	177.747
22	nos7.mtx	729	2673	12.6713	289.779	13.8171	51.748	16.3483	138.696
23	bcsstk19.mtx	817	3835	33.5146	19744.8	47.8738	4113.49	35.7862	4755.67
24	gr_30_30.mtx	900	4322	12.4014	4.96459	7.05082	0.972827	11.4049	4.66882
25	nos2.mtx	957	2547	10.2019	3859.65	5.53684	645.577	10.3288	57.8885
26	1138_bus.mtx	1138	2596	24.8954	253.147	17.4776	38.3676	23.9189	1640.66
27	bcsstm26.mtx	1922	1922	30.1713	347.351	21.9322	34.4225	32.2852	11393.8
28	Chem97ZtZ.mtx	2541	4951	63.1019	21.173	29.6964	2.7269	40.5421	976.889
29	bcsstm23.mtx	3134	3134	46.6313	1174.33	32.0554	133.187	48.4448	70434.9
30	bcsstm24.mtx	3562	3562	52.4761	6732.79	36.3927	689.691	55.634	403048
31	bcsstm21.mtx	3600	3600	58.3939	0.935506	39.4897	0.295145	63.6318	48.5527
32	t2dal_e.mtx	4257	4257	81.0908	5049.31	64.5586	593.571	84.607	367754

8.3 დიდი მატრიცები

				Boost Vectors		CG Sparse		Vectors	
#		N	NNZ	Fill	Solve	Fill	Solve	Fill	Solve
1	bcsstk34.mtx	588	11003	160.706	127.479	204.149	41.8899	134.18	72.9352
2	Trefethen_700.mtx	700	6677	31.9672	26.1957	68.456	6.99683	19.8414	209.681
3	msc00726.mtx	726	17622	317.378	262.281	292.801	96.4214	258.399	103.019
4	ex33.mtx	1733	11961	197.702	23490	226.516	5174.58	235.654	6046.14
5	ex3.mtx	1821	27253	466.813	6263.29	455.707	1767	431.021	1231.69
6	nasa1824.mtx	1824	20516	371.59	2011.37	386.585	613.43	338.41	5485.98
7	nasa2146.mtx	2146	37198	577.162	155.461	802.113	55.5489	519.302	717.563
8	ex10.mtx	2410	28625	590.405	24614.2	594.31	8728.89	455.914	74521.6
9	ex10hs.mtx	2548	29928	538.325	25669.3	531.984	8009.36	536.798	64499.5
10	ex13.mtx	2568	39098	702.745	286848	588.736	105880	559.92	268135
11	nasa2910.mtx	2910	88603	2636.99	8000.05	1523.56	3310.32	1458.94	20204.4
12	ex9.mtx	3363	51417	911.301	247547	815.523	79170.9	761.973	91656.4
13	sts4098.mtx	4098	38227	1450.06	27826.7	806.845	8022.19	701.543	648469
14	msc04515.mtx	4515	51111	717.055	4076.68	766.662	1342.46	753.42	3072.34
15	nasa4704.mtx	4704	54730	1029.52	23355.3	914.46	7501.1	861.621	118858
16	Muu.mtx	7102	88618	2780.3	93.8574	1723.98	27.6281	1798.71	2077.48
17	bcsstk38.mtx	8032	181746	4579.15	2922350	2078.07	1507550	2174.73	13787700
18	bloweybq.mtx	10001	39996	848.226	19520300	89.8954	3130290	160.44	1582.21
19	bundle1.mtx	10581	390741	164973	827.949	1093.09	346.603	1363.15	72995.2

ლიტერატურა

1. E. D. Dolan and J. J. Mor´e, Benchmarking optimization software with performance profiles, Math. Program., 91 (2002), 201–213.
2. Davis, Y. Hu, The University of Florida Sparse Matrix Collection, ACM Transactions on Mathematical Software, Vol. V, No. N, M 20YY, 1–28.
3. boost library. See http://www.boost.org/doc/libs/1_60_0/libs/numeric/ublas/doc/matrix_sparse.html
4. Linear Algebra Libraries. See http://verdandi.sourceforge.net/doc/linear_algebra_libraries.pdf
5. Yousef Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, 2003, SIAM
6. G. Gundersen, T. Steihaug, Data structures in Java for matrix computations, Concurrency and Computation: Practice and Experience, 2004, 799-815
7. LAPACK — Linear Algebra PACKage. See http://www.netlib.org/lapack/
8. The University of Florida Sparse Matrix Collection. See https://www.cise.ufl.edu/research/sparse/matrices/
9. https://en.wikipedia.org/wiki/Google_matrix
10. https://books.google.pl/books?id=KsHTl_2Pfl8C&pg=PA40&lpg=PA40&dq=google+sparse+matrix&source=bl&ots=rMmY0xa4UB&sig=sw4jG5xlt2yTs_50bNXPqamhRnY&hl=en&sa=X&ved=0ahUKEwiHkPP4t8rUAhUrIpoKHRKWB2oQ6AEISzAG#v=onepage&q=google%20sparse%20matrix&f=false