Matrius Disperses

Marc Cané Ismael El Habri Lluís Trilla

7 de novembre de 2018

Table of Contents

Què són les matrius disperses?

Quan parlem de matrius disperses ens referim a matrius de gran tamany en la qual la majoria d'elements son zero. Direm que una matriu és dispersa, quan hi hagi benefici en aplicar els mètodes propis d'aquestes.

Per identificar si una matriu és dispersa, podem usar el següent:
Una matriu $n \times n$ serà dispersa si el número de coeficients no nuls es $n^{\gamma+1}$, on $\gamma < 1$.
En funció del problema, decidim el valor del paràmetre γ. Aquí hi ha els valors típics de γ:

- $\gamma = 0.2$ per problemes d'anàlisi de sistemes elèctrics de generació i de transport d'energia.
- $\gamma = 0.5$ per matrius en bandes associades a problemes d'anàlisi d'estructures.

Tipus de matrius disperses

Podem trobar dos tipus de matrius disperses:

- **Matrius estructurades:** matrius en les quals els elements diferents de zero formen un patró regular. Exemple: Les matrius banda.
- **Matrius no estructurades:** els elements diferents de zero es distribueixen de forma irregular.

Table of Contents

Per Coordenades

És la primera aproximació que podríem pensar i és bastant intuïtiva. Per cada element no nul guardem una tupla amb el valor i les seves coordenades: (a_{ij}, i, j).
A la realitat però, aquest mètode d'emmagatzemar les dades és poc eficient quan hem de fer operacions amb les matrius.

Exemple

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & -2 & 0 \end{pmatrix} =$$

índex	tupla(a_{ij}, i, j)
0	$(1, 1, 1)$
1	$(2, 1, 4)$
2	$(1, 2, 2)$
3	$(3, 4, 1)$
4	$(-2, 4, 3)$

Usem tres vectors de la mateixa mida (n_z, el nombre d'elements diferents de zero): Un amb els valors, un amb les files i un amb les columnes:

Vector	Coeficients				
valors	1	2	1	3	-2
files	1	1	2	4	4
columnes	1	4	2	1	3

Per Files

També conegut com a *Compressed Sparse Rows (CSR)*, *Compressed Row Storage (CRS)*, o format *Yale*. És el mètode més estès.
Consisteix en guardar els elements ordenats per files, guardar la columna on es troben, i la posició del primer element de cada fila en el vector de valors. Així ens quedaran tres vectors:

- **valors:** de mida n_z, conté tots els valors diferents.
- **columnes:** també de mida n_z, conté la columna on es troba cada un dels elements anteriors.
- **iniFiles:** de mida $m+1$, conté la posició on comença cada fila en els vectors valors i columnes, sent m el nombre de files de la matriu.

Exemple

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & -2 & 0 \end{pmatrix} =$$

Vector	Coeficients				
índex	1	2	3	4	5
valors	1	2	1	3	-2
columnes	1	4	2	1	3
iniFiles	1	3	4	4	6

Si es canvien files per columnes, dona la implementació per columnes, o també anomenada *Compressed Sparse Columns (CSC)*.

Per Perfil

Aquest mètode és una manera eficient de guardar un tipus concret de matrius, les matrius banda.

Matrius banda

Com vam veure a classe, una matriu $n \times n$ és banda si existeixen p i q naturals, tals que $1 < p, q < n$ i $a_{ij} = 0$ sempre que $p \leq j - i$ o $q \leq i - j$.
Anomenem ample de banda de la matriu a $p + q - 1$.
L'**envoltant** de una matriu banda consisteix en tots els elements de cada fila des de el primer no nul fins al ultim no nul, incloent els elements nuls que hi pugui haver entre mig. Aquí la definició formal:
L'envoltant de una matriu banda A, $env(A)$ es defineix com el conjunt
$env(A) = \{i, j\} : f_i \leq j \leq l_i, 1 \leq i \leq n$ on f_i és on comencen els valors no nuls de la fila i, i l_i on acaben els valors no nuls de la fila i.

El mètode

Consisteix en guardar els elements de l'envoltant de la matriu, la columna on comença l'envoltant en cada fila, i la posició en el vector de valors on comença cada fila.
Doncs, ens quedarien tres vectors:

- **valors:** amb els valors de l'envoltant.
- **columnInici:** amb el valor f_i de cada fila (la columna on comença l'envoltant en cada fila).
- **iniFiles:** amb la posició on comença cada fila en el vector valors.

Exemple

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & -2 \end{pmatrix} =$$

Vector	Coeficients						
índex	1	2	3	4	5	6	7
valors	1	2	1	2	0	2	−2
columnInici	1	2	2	4			
iniFiles	1	3	4	7			

Altres mètodes

- **Diccionari de claus:** Consisteix en mantenir un diccionari que mapeja parelles de (fila, columna) amb el valor de cada element no nul.
- **Llista de llistes:** Guarda una llista per fila, en la qual cada entrada és una parella de valors (valor, columna).
- **Esquema DIA:** Aquest esquema s'usa quan els valors no nuls estan restringits a un nombre reduït de diagonals. Consisteix en guardar una matriu de dades que conté els valors no nuls i un vector amb els *offsets*, que guarda el desplaçament de cada diagonal respecte la diagonal principal.

Implementació amb Matlab del mètode CSR

Hem implementat un script Matlab amb una classe `CSRSparseMatrix` que guardi les dades necessàries. Aquestes les tenim en "l'atribut" `Matrix` dins del bloc `properties` (línia 10 del codi següent). Aquestes dades consisteixen en el següent:

- `Matrix.nColumns`: número de columnes de la matriu, necessari per recrear les files posteriorment.
- `Matrix.values`: vector valors comentat anteriorment, amb els valors no nuls de la matriu.
- `Matrix.columns`: vector de columnes, amb la columna corresponent a cada valor amb el mateix índex.
- `Matrix.beginningRow`: vector amb els índex comença cada fila en el vector de valors i de columnes.

```
%===============CSRSparseMatrix=================================================
% Classe que implementa el mètode d'emmagatzematge per files sobre matrius
%%% disperses
classdef CSRSparseMatrix
  properties
    Matrix
    %%% La matriu (estructura de quatre elements:
    %%%                n: Matrix.nColumns,
    %%%                valors: Matrix.values,
    %%%                columnes: Matrix.columns,
    %%%                inici files: Matrix.beginningRow)
  end

  methods
  %
  %%% MÈTODES
  %
  end
end
```

Constructor

Entenent com s'emmagatzema la matriu, fer el constructor de la classe és trivial: només és necessari recórrer la matriu per files i guardar al vector de valors els elements diferents de 0, guardar-nos la columna on es troba cada un d'aquests elements en el vector de columnes. A part, per cada fila, hem de actualitzar el vector d'índexs de files. Farem el següent:

- `Matrix.beginningRow[1] = 1`.
- `Matrix.beginningRow[i] = Matrix.beginningRow[i-1]` + elements diferents de zero en la fila i.

```
%==============Constructor=============================================================
%
% El constructor rep una matriu i hi aplica el metode, tornant un objecte
%%% de tipus CSRSparseMatrix.
%
function obj = CSRSparseMatrix(A)
  [m,n] = size(A);
  obj.Matrix.nColumns = n;
  obj.Matrix.values = [];
  obj.Matrix.columns = [];
  obj.Matrix.beginningRow = [1];
  for i = 1:m
    nonZero=obj.Matrix.beginningRow(i);
    nonZeroThisRow=0;
    for j = 1:n
      if(A(i,j) ~= 0)
        obj.Matrix.values = [obj.Matrix.values, A(i,j)];
        obj.Matrix.columns = [obj.Matrix.columns, j];
        nonZeroThisRow=nonZeroThisRow+1;
      end
    end
    obj.Matrix.beginningRow = [obj.Matrix.beginningRow, nonZero+nonZeroThisRow];
  end
end
```

Mètodes per obtenir una fila, una columna i un element

El mètode per obtenir una fila es troba a partir de la línia 6 del codi posterior. Com podem veure consisteix en primer crear una fila amb `Matrix.nColumns` zeros. Llavors, sabent que `Matrix.beginningRow[i]` ens indica on comença la fila i en el vector de valors i en el de columnes, i deduint que acaben a `Matrix.beginningRow[i+1]-1`; podem delimitar els elements de la fila. Només quedaria afegir aquests elements a les seves posicions corresponents (usant el vector de columnes).
El mètode per obtenir un element es troba a partir de la línia 22 del codi posterior. Aquest mètode, usa el que hem vist anteriorment per crear la fila determinada, llavors només cal obtenir l'element amb la columna.
El mètode per obtenir una columna es troba a partir de la línia 32 del codi posterior. Sabent quantes files tenim (recordem que `Matrix.beginningRow` té $m+1$ elements, on m és el nombre de files de la matriu), inicialitzem cada element a 0 i busquem els elements de cada posició de la columna, usant el mateix mètode usat per obtenir les files. Cal tenir en compte, que per cada fila, les columnes estan ordenades, així que no cal buscar més quan ja hem passat la columna que estem mirant.

```
%============getRow===============================================================
%
%%% Donats obj, sent el objecte actual, i x la fila que volem obtenir;
%%%% retorna la fila x de la matriu obj
%
function row = getRow(obj, x)
  row = 0;
  for i= 1:obj.Matrix.nColumns
    row(i)=0;
  end
  for i = obj.Matrix.beginningRow(x):obj.Matrix.beginningRow(x+1)-1
    row(obj.Matrix.columns(i)) = obj.Matrix.values(i);
  end
end
```

```
%============getElem=================================================================
%
%%% Donats obj, sent el objecte actual i (x,y) les coordenades del element
%%%% que volem obtenir;
%%%% retorna l'element x,y de la matriu obj
%
function elem = getElem(obj, x, y)
  row = obj.getRow(x);
  elem = row(y);
end
```

```
%============getColumn=================================================================
%
%%% Donats obj, sent el objecte actual, i y la columna que volem obtenir;
%%%% retorna la columna y de la matriu obj
%
function column = getColumn(obj, y)
  column = 0;
  m=size(obj.Matrix.beginningRow,2)-1;
  for j=1:m
    column(j)=0;
    for i=obj.Matrix.beginningRow(j):obj.Matrix.beginningRow(j+1)-1
      if obj.Matrix.columns(i)==y
          column(j) = obj.Matrix.values(i);
          break
      elseif obj.Matrix.columns(i)> y
          break
      end
    end
  end
  column = transpose(column);
end
```

Table of Contents

Mètode per afegir una fila a una matriu dispersa

Aquest mètode està pensat per construir les matrius de forma incremental, cosa que ens serà molt útil per fer les operacions posteriors.
El mètode en sí consisteix en fer exactament el mateix que fa el constructor per cada fila, per la donada per paràmetre en el mètode.

```
%============addRow===============================================================
%
%%% Donats obj, sent el objecte actual, i row la fila que volem afegir;
%%%% retorna obj amb la fila nova afegida
%
function obj = addRow(obj, row)
  i=size(obj.Matrix.beginningRow,2);
  nonZero=obj.Matrix.beginningRow(i);
  nonZeroThisRow=0;
  [_,n] = size(row);
  for j = 1:n
    if(row(j) ~= 0)
      obj.Matrix.values = [obj.Matrix.values, row(j)];
      obj.Matrix.columns = [obj.Matrix.columns, j];
      nonZeroThisRow=nonZeroThisRow+1;
    end
  end
  obj.Matrix.beginningRow = [obj.Matrix.beginningRow, nonZero+nonZeroThisRow];
end
```

Mètode per sumar dues matrius disperses

Per sumar dues matrius disperses sense carregar tota la matriu en memòria sumem fila per fila i ho afegim a una nova matriu dispersa resultat.

```
%============sum=================================================================
%
%%% Donats obj i una altre matriu dispersa de la mateixa mida;
%%%% retorna la suma de les dues matrius
%
function res = plus(obj, B)
  nRowsA = length(obj.Matrix.beginningRow)-1;
  nRowsB = length(B.Matrix.beginningRow)-1;
  assert(obj.Matrix.nColumns == B.Matrix.nColumns && nRowsA == nRowsB)
  res = CSRSparseMatrix([]);
  res.Matrix.nColumns = obj.Matrix.nColumns;
  for i=1:nRowsB
    rowA = obj.getRow(i);
    rowB = B.getRow(i);
    res = res.addRow(rowA+rowB);
  end
end
```

Mètodes pel producte matriu-vector

Aquest apartat es dividirà en dos mètodes:

- El producte d'un vector per una matriu.
- El producte d'una matriu per un vector.

Producte d'un vector per una matriu

Consisteix en la multiplicació $c = b \times A$, on A és una matriu $m \times n$, b és un vector de mida m i c és un vector de mida n.
El resultat d'aquesta operació, es pot representar així, pel cas que $m = 2$ i $n = 3$:

$$\begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix} = \begin{bmatrix} b_1 & b_2 \end{bmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \quad \longrightarrow \quad \begin{array}{l} c_1 = b_1 a_{11} + b_2 a_{21} \\ c_2 = b_1 a_{12} + b_2 a_{22} \\ c_3 = b_1 a_{13} + b_2 a_{23} \end{array}$$

Això es podria generalitzar de la següent manera:

$$c_j = \sum_{i=1}^{m} b_i a_{ij}$$

Amb aquesta formula, la implementació es simplifica bastant: només cal aplicar-la per a cada columna j, quedant un doble bucle, com es pot veure en el codi a continuació:

```
%============multRow============================================================
%
%%% Donats obj i un vector de la mateixa mida que les columnes de obj
%%%% retorna el resultat de la multiplicació b*obj
%
function res = multRow(obj,b)
  res = zeros(1, length(obj.Matrix.beginningRow)-1);
  assert(obj.Matrix.nColumns == length(b));
  for i=1:obj.Matrix.nColumns
    bi = b(i);
    for ii = obj.Matrix.beginningRow(i):obj.Matrix.beginningRow(i+1)-1
      j = obj.Matrix.columns(ii);
      res(j) = res(j) + obj.Matrix.values(ii)*bi;
    end
  end
end
```

Producte d'una matriu per un vector

Consisteix en la multiplicació $c = A \times b$, on A és una matriu $m \times n$, b és un vector de mida n i c és un vector de mida m.
En aquest cas el que fem és cada fila i de la matriu, i la multipliquem per b. El resultat el guardem a c_i.

```
%============multColumn============================================================
%
%%% Donats obj i un vector de la mateixa mida que les files de obj
%%%% retorna el resultat de la multiplicació obj*b
%
function res = multColumn(obj, b)
  res=zeros(1, length(obj.Matrix.nColumns));
  m = length(obj.Matrix.beginningRow)-1;
  assert ( m == length(b));
  for i = 1:m
    row = obj.getRow(i);
    res(i)= row * b;
  end
  res = transpose(res);
end
```

Mètode per multiplicar dues matrius

Consisteix en la multiplicació de $C = A \times B$, on A és una matriu $m \times n$ i B és una matriu $n \times o$, resultant en la matriu C que és de mida $m \times o$.
Aquesta operació la podem representar simbòlicament de la següent manera per a $m = 2, n = 3, o = 2$:

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} \quad \longrightarrow \quad \begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{aligned}$$

Amb això podem veure com hi ha un patró semblant al vist en la multiplicació vector per matriu. Cada fila i de C és igual a la multiplicació de la fila i de A per la matriu B. Per tant podem fer que en la implementació, per cada fila i de la matriu C, obtenir la fila i de A i fer $A_i \times B$ usant el mètode implementat anteriorment.

```
%============mtimes=================================================================
%
%%% Donats obj de mida m*n i una matriu dispersa B de mida n*o,
%%%% retorna el resultat de la multiplicació obj*B de mida m*o
%
function res = mtimes(obj, B)
  nRowsA = length(obj.Matrix.beginningRow)-1;
  nRowsB = length(B.Matrix.beginningRow)-1;
  assert(obj.Matrix.nColumns == nRowsB)
  res = CSRSparseMatrix([]);
  res.Matrix.nColumns = B.Matrix.nColumns;

  for i=1:nRowsB
    row = obj.getRow(i);
    res = res.addRow(B.multRow(row));
  end
end
```

Table of Contents

Introducció

Per resoldre un sistema d'equacions representat amb una matriu dispersa podríem usar qualsevol dels mètodes que coneixem per resoldre matrius, però ens trobaríem fent un nombre molt gran d'operacions innecessàries i gastant molta més memòria de la necessària.

Suposem que apliquem el mètode de Gauss en una matriu dispersa, podem veure marcats en vermell tots els elements no nuls que hem introduït al aplicar Gauss (elements *fill-in*):

$$\begin{pmatrix} x & 0 & x & 0 & 0 \\ 0 & x & 0 & 0 & x \\ x & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ 0 & x & 0 & 0 & x \end{pmatrix} \xrightarrow{\text{Gauss}} \begin{pmatrix} x & 0 & x & 0 & 0 \\ 0 & x & x & 0 & x \\ x & x & x & 0 & x \\ 0 & 0 & 0 & x & x \\ 0 & x & x & x & x \end{pmatrix}$$

Això ens suposaria un increment de l'ús de memòria i ens obligaria a incrementar molt el número d'operacions, augmentant la complexitat numèrica. Així doncs, ens interessa reordenar files i columnes per tal de reduir o fins i tot desfer-nos dels elements de *fill-in*.

Per resoldre qualsevol sistema d'equacions representat amb una matriu dispersa, seguirem els següents passos:

1. Determinar l'estructura simbòlica de A.
2. Obtenir les permutacions P i Q, tals que PAQ tingui una estructura de densitat òptima (no genera elements de *fill-in*).
3. Fer factorització LU: $LU = PAQ$ i $c = Pb$.
4. Resoldre $Lz = c$, $Uy = z$ i $x = Qy$.

Determinació de l'estructura simbòlica

Per poder fer el següent apartat, la reordenació, primer cal saber com és la matriu que tenim, ja que en funció d'això requerirem d'un algoritme diferent. Per exemple, amb matrius simètriques, hi ha un gran nombre de algoritmes per fer la reordenació més eficientment. En canvi per matrius no simètriques, els algoritmes necessaris són uns altres.
Per saber-ho fem una representació simbòlica de com és la matriu d'equacions.

Mètodes de reordenació

Com hem vist amb anterioritat, reordenar les matrius de forma correcta és de vital importància, presentant tres millores fonamentals:

- La disminució de la memòria necessària per emmagatzemar la matriu generada en el procés de factorització.
- La disminució del nombre de càlculs a realitzar, reduint el temps total de càlcul.
- La millora de l'estabilitat numèrica del procés global de resolució del sistema al disminuir el nombre de coeficient a considerar, reduint així la possible acumulació d'error en la resolució del sistema.

Per aconseguir això tenim diferents mètodes, en funció de l'estructura simbòlica de la matriu sobre la que estem treballant, i de com aquesta estigui emmagatzemada.
En el cas que la matriu tingui una estructura simètrica i s'emmagatzemi amb un esquema de perfil, també ens interessarà una ordenació que agrupi els elements a prop de la diagonal principal. Per aquest propòsit existeix l'**algoritme de Cuthill-Mckee**.
Altres mètodes per matrius d'estructura simètrica són: l'**eliminació de Gauss mitjançant grafs** i l'**algoritme del grau mínim**.
Per ordenar matrius disperses d'estructura no simètrica existeixen diversos mètodes, entre ells el més utilitzat és l'ordenació amb estructura triangular inferior en blocs.

Representació d'una matriu amb grafs

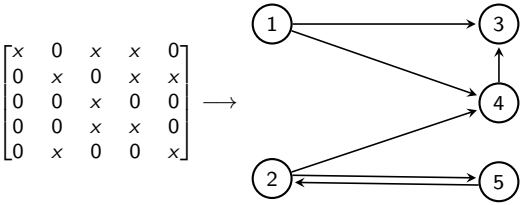
Com ja hem vist a classe en anys anteriors, un graf és un parell format per un conjunt finit de vèrtexs i un altre conjunt finit d'arestes. Es podria representar així: $G = (V, A)$, on V és el conjunt de vèrtexs i A el conjunt d'arestes.
Una aresta està formada per dos vèrtexs. Si les arestes d'un graf estan ordenades en direm digraf o graf dirigit. Si no, en direm graf no dirigit, o graf a seques.
El grau d'un vèrtex és el numero de vèrtexs veïns que té, és a dir, el nombre de vèrtexs units a ell. En un graf dirigit, podem separar-lo en grau d'entrada i grau de sortida, sent aquests el nombre d'arestes que en surten i el nombre d'arestes que hi arriben respectivament.
A qualsevol matriu quadrada $n \times n$ se li pot associar un digraf amb n nodes.
En aquest graf $G^A = (V^A, E^A)$, les arestes E^A són tals que:

$$(v_i, v_j) \in E^A \Leftrightarrow a_{ij} \neq 0$$

A qualsevol coeficient diagonal a_{ii} també li correspon una aresta circular en el vèrtex, en cas que tots els elements de la diagonal fossin no nuls, aquestes arestes no es representen.
Propietat important: Si a una matriu se li efectuen permutacions simètriques, en el seu digraf associat només es modifica l'enumeració dels nodes.

Representació d'una matriu amb grafs

Exemple



Reordenació en matrius disperses no simètriques

Si al sistema d'equacions $Ax = b$ se li apliquen unes permutacions a l'esquerra i a la dreta representades per les matrius P i Q, el sistema original es reescriu:

$$PAQQ^T x = Pb \qquad \text{on } QQ^T = I.$$

Podem simplificar-ho com a:

$$By = c \qquad \text{on } B = PAQ \text{ és la matriu } A \text{ reordenada.}$$

Si A no té cap estructura especial, sempre es pot reordenar de forma triangular en blocs :

$$\begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & & \ddots & \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

on els A_{ij} son matrius, sent les A_{ii} quadrades de ordre n_i: $\sum_{i=1}^{n} n_i = n$

Eliminació de Gauss en matrius generals disperses

Si el sistema no presenta una estructura de dispersió especial, i tampoc ens interessa triangularitzar-la per blocs, la forma més general de resoldre-la consisteix en utilitzar directament l'eliminació de Gauss tenint en compte que la matriu és dispersa.
Per fer-ho s'usa el criteri de Markowitz, de Harry Max Markowitz.
Consisteix en factoritzar la matriu mitjançant l'eliminació de Gauss amb pivotatge, triant com a element pivot en una etapa k un a_{ij}^k numèricament acceptable i que minimitzi $(r_i^k - 1)(c_i^k - 1)$, on r_i^k és el numero de coeficients no zero en la fila i de la sub-matriu activa i c_i^k els no nuls en la columna j.

Estructura triangular en blocs

El procediment per triangular per blocs una matriu A consisteix en dues fases:

1. Trobar la transversal completa d'aquesta matriu.
2. Reordenar el resultat del pas anterior utilitzant permutacions simètriques.

Qualsevol matriu A es pot reordenar de tal forma que PAQ tingui una transversal completa. Si la matriu és singular (el seu determinant és 0) això pot no ser cert.
Amb la transversal completa, pot reordenar-se la matriu utilitzant permutacions simètriques per aconseguir una estructura triangular inferior en blocs.

- Si per una matriu A existeix una estructura triangular inferior en blocs, en direm matriu reductible.
- Si una matriu no té transversal completa, però pot reordenar-se de tal forma que en tingui i, un cop reordenada, continuï sent reductible, en direm birreductible.

Transversal completa

Per trobar una matriu B a partir de permutar files i columnes de A, de forma que la matriu obtinguda tingui transversal completa, usarem l'algoritme de Hall.
L'algoritme de Hall requereix de n iteracions per una matriu de mida $n \times n$ en les quals, mira si el coeficient corresponent de la diagonal és diferent de zero, i si no ho és, hi col·loca un element diferent de zero.
Estant en la iteració $it = k + 1$, per tant s'han realitzat k iteracions, i els k primers elements de la diagonal sabem que ja són diferents de zero.

Transversal completa

Algoritme

En la iteració it, es poden donar 3 casos:

- $a_{itit} \neq 0$: no s'ha de fer res.
- $a_{itit} = 0$ i existeixen coeficients no nuls en la sub-matriu activa[1]
  - Fem els intercanvis de files i columnes necessaris per portar un d'aquests coeficients a la posició (it, it).
- Que només existeixin coeficients no nuls en la sub-matriu activa[1]
  - Mitjançant el mètode del camí creixent es pot aconseguir col·locar un coeficient diferent de zero a a_{itit}. Si es donés el cas que no existís aquest camí, voldria dir que la matriu és singular.

[1] La sub-matriu quadrada formada per la diagonal des de (it, it) fins a (n, n).

Traçat del camí creixent

En la iteració it:

- Recorrem la fila it fins a trobar un coeficient no nul, en una columna l (tal columna ha d'existir, sinó voldria dir que tots els coeficients de la fila it són 0, i per tant la matriu seria singular).
- Ens desplacem a la posició (l, l) i tornem a aplicar el pas anterior des del nou punt.

L'algoritme acabarà en un coeficient diferent de zero en la sub-matriu definida per les files de 1 a k i les columnes $k+1$ a n.
Propietat: Aquest camí no pot passar per una fila o columna més d'un cop i per tant, tampoc un mateix coeficient de la diagonal.

Obtenció de l'estructura triangular inferior per blocs

Havent obtingut una matriu amb transversal completa, la següent fase per arribar a una matriu triangular en blocs consisteix en trobar una altre permutació que ens generi

$$\begin{pmatrix} B_{11} & & & \\ B_{21} & B_{22} & & \\ \vdots & & \ddots & \\ B_{n1} & B_{n2} & \dots & B_{nn} \end{pmatrix}$$

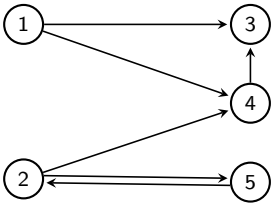
on cada bloc diagonal B_{ii} no es pugui reduïr més.
Els algoritmes per aconseguir això es basen en trobar cicles fortament connexos[2] en el digraf associat. Si existissin dos cicles fortament connexos i re-enumeréssim els vèrtexs de tal forma que els del primer cicle fossin els vèrtexs de 1 a k i els del segon de $k+1$ fins a n, obtindríem una matriu triangular inferior en dos blocs, el primer amb k columnes, i el segon amb $n-k$.

[2] Eren aquells cicles en els quals podies arribar a qualsevol vertex del cicle des d'un d'aquests.

Obtenció de l'estructura triangular inferior per blocs

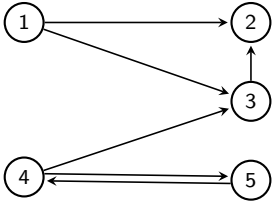
Exemple

$$A = \begin{bmatrix} x & 0 & x & x & 0 \\ 0 & x & 0 & x & x \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & x & x & 0 \\ 0 & x & 0 & 0 & x \end{bmatrix} \longrightarrow G_A =$$



Com veiem G_A té dos cicles fortament connexos, el primer format, per $(1, 3, 4)$ i el segon format per $(2, 5)$. Re-enumerant $(1, 3, 4)$ per $(1, 2, 3)$ i $(2, 5)$ per $(4, 5)$, ens donaria el següent graf G_B representant a la matriu B:

$$G_B =$$



$$\longrightarrow B = \begin{bmatrix} x & x & x & 0 & 0 \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{bmatrix}$$

Table of Contents

Factorització LU de matrius

La factorització LU és el procés de descomposar una matriu A en dues matrius L i U tal que $A = LU$ on L sigui una matriu trinagular inferior i U una matriu triangular superior.
Utilitzar el mètode de Gauss per resoldre el sistema d'equacions tindria un ordre de complexitat de $O(n^3)$, mentre que amb la factorització LU tenim un problema d'ordre $O(n^2)$ un cop aconseguides les matrius LU. En aquest cost, però, no estem tinguent en compte el cost d'obtenir les matrius L i U, i per tant, hem de tenir en compte que amb matrius de petita mida ens podria ser més ràpid seguir usant altres mètodes. És només quan tenim matrius de gran tamany, i hem de resoldre el sistema d'equacions per a diverses b que ens sortirà a compte fer el procés de factorització.

Factorització LU de matrius

Usos

Donat que $LU = A$, podriem definir l'equació $LUx = b$ on x és la solució al sistema, i b és el vector de termes independents.
La factorització LU s'utilitza molt sovint en computació per solucionar repetidament sistemes d'equacions amb diferents termes independents. Una de les altres aplicacions és la de trobar el determinant de la matriu original A. Donat que $A = LU$, podem assumir que $det(A) = det(LU) = det(L) * det(U)$. Aprofitant els 0s de cada matriu triangular podem trobar de manera molt més ràpida els seus determinants.

Table of Contents

Estalvi d'espai

Representar una matriu en forma dispersa ens permetrà estalviar espai de memòria respecte a la seva versió densa. Podríem dir que l'espai de memòria que ocupa una matriu densa seria donat per la formula $E = C^n$ on E seria el nombre d'elements a guardar, C seria el tamany del costat, i n seria el seu número de dimensions. De la mateixa forma, podríem definir el tamany mínim que ocupa una matriu dispersa com a $E = (C^n) - X$ on E seria el nombre d'elements a guardar, C seria el tamany del costat, n seria el seu número de dimensions i X seria el nombre d'elements de la matriu iguals a zero.

Estalvi d'espai

Exemples

Per exemple, amb una matriu diagonal de 10000×10000, en forma densa tindria 100000000 elements, i en forma dispersa podria arribar a ocupar només 10000 elements.
Utilitzant el mètode de representació per coordenades, podríem calcular el nombre d'elements a guardar amb la formula $E = ((C^n) - X) * (n + 1)$. Per exemple, amb una matriu diagonal de 10000×10000, utilitzant la representació per coordenades ocuparia tant sols 30000 elements.
Utilitzant el mètode de representació per files, podríem calcular el nombre d'elements a guardar en una matriu quadrada amb la formula $E = ((C^2) - X) * 2 + C + 1$. Per exemple, amb una matriu diagonal de 10000×10000, utilitzant la representació per coordenades ocuparia 30001 elements. No obstant, si la matriu tingués més elements escalaria millor que usant el mètode de coordenades.
Utilitzant el mètode de representació per perfil, podríem calcular el nombre d'elements a guardar en una matriu quadrada amb la formula $E = ((C^2) - X) + numIndexos + C + 1$. Per exemple, amb una matriu diagonal de 10000×10000, utilitzant la representació per coordenades ocuparia $20001 + numIndexos$ elements. Com ja hem explicat anteriorment, aquest mètode depèn de la quantitat de valors a prop de la diagonal, i és molt útil per a matrius banda.

Estalvi de temps

Per tal de mesurar i comparar les diferències de rendiment entre la nostre implementació de matrius disperses i la implementació de matrius denses del matlab, crearem una matriu de 1000×1000 elements, on al voltant d'un 95% dels elements seran buits.
Un cop creada la matriu procedirem a mesurar el temps que tarden les operacions de suma i multiplicació amb una altre matriu d'iguals característiques. Per tal de reduir la variança dels nostres resultats, hem executat cada operació 10 cops seguits i n'hem fet la mitjana aritmètica, als seguents apartats tenim els resultats obtinguts.

Estalvi de temps

Resultats

	Iteracions	Temps total	Temps per iteració
Implementació dispersa	10	104.52s	10.45s
Matlab densa	10000	10.37s	1.03ms
Matlab sparse	10000	3.88s	0.38ms

	Iteracions	Temps total	Temps per iteració
Implementació dispersa	1	1873s	1873s
Matlab densa	100	64.62s	646.2ms
Matlab sparse	1000	25.72s	25.72ms

Estalvi de temps

Conclusions

Veient els temps per iteració podem observar que la nostre implementació segueix de matrius disperses és inclús més lenta que la implementació de matrius denses de matlab. En la suma de matrius ha tardat 10000 cops més per iteració, i en la multiplicació ha costat 31 minuts fer una sola iteració. Tot i que no hem trobat els detalls de la implementació que usa matlab, podem assumir que la implementació interna de les matrius a matlab està feta en un llenguatge a nivell baix, tipus C o C++, enlloc d'usar el propi llenguatge de matlab.
També podria ser que usessin instruccions SIMD per tal de millorar el rendiment.
Tot i així, observant els resultats de la propia implementació de matrius disperses de Matlab podem veure que és molt més eficient que tractant la matriu com a una matriu normal. En la suma de matrius veiem una millora del 271%, però la diferència més important s'observa en la multiplicació, on podem veure una diferència del 2500%.