sparse matrices and graphs

L. Olson

Department of Computer Science
University of Illinois at Urbana-Champaign
objectives

- Convert a graph into a *sparse* matrix
- Go over a few sparse matrix storage formats
- Give an example of lower memory benefits
- Give an example of computational complexity benefits
sparse matrices

- Vague definition: matrix with few nonzero entries
- For all practical purposes: an $m \times n$ matrix is sparse if it has $\Theta(\min(m, n))$ nonzero entries.
- This means roughly a constant number of nonzero entries per row and column
sparse matrices

- Other definitions use a slow growth of nonzero entries with respect to $n$ or $m$.
- Wilkinson’s Definition: “..matrices that allow special techniques to take advantage of the large number of zero elements.” (J. Wilkinson)
- A few applications which lead to sparse matrices: Structural Engineering, Computational Fluid Dynamics, Reservoir simulation, Electrical Networks, optimization, data analysis, information retrieval (LSI), circuit simulation, device simulation, ...
sparse matrices: the goal

- To perform standard matrix computations economically i.e., without storing the zeros of the matrix.
- For typical Finite Element /Finite difference matrices, number of nonzero elements is $\Theta(n)$.

Example

To add two square dense matrices of size $n$ requires $\Theta(n^2)$ operations. To add two sparse matrices $A$ and $B$ requires $\Theta(nnz(A) + nnz(B))$ where $nnz(X)$ = number of nonzero elements of a matrix $X$.

remark

$A^{-1}$ is usually dense, but $L$ and $U$ in the $LU$ factorization may be reasonably sparse (if a good technique is used).
goal

- Principle goal: solve

\[ Ax = b \]

where \( A \in \mathbb{R}^{n \times n}, x, b \in \mathbb{R}^n \)

- Assumption: \( A \) is very sparse

- General approach: iteratively improve the solution

- Given \( x_0 \), ultimate “correction” is

\[ x_1 = x_0 + e_0 \]

where \( e_0 = x - x_0 \), thus \( Ae_0 = Ax - Ax_0 \),

- or

\[ x_1 = x_0 + A^{-1}r_0 \]

where \( r_0 = b - Ax_0 \)
• Principle difficulty: how do we “approximate” $A^{-1}r$ or reformulate the iteration?
• One simple idea:

$$x_1 = x_0 + \alpha r_0$$

• operation is inexpensive if $r_0$ is inexpensive
• requires very fast sparse mat-vec (matrix-vector multiply) $Ax_0$
.sparse matrices

- So how do we store $A$?
- Fast mat-vec is certainly important; also ask
  - what type of access (rows, cols, diag, etc)?
  - dynamic allocation?
  - transpose needed?
  - inherent structure?
- Unlike dense methods, not a lot of standards for iterative
  - dense BLAS have been long accepted
  - sparse BLAS still iterating
- Even data structures for dense storage not as obvious
- Sparse operations have low operation/memory reference ratio
### popular storage structures

<table>
<thead>
<tr>
<th>DNS</th>
<th>Dense</th>
<th>ELL</th>
<th>Ellpack-Itpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>BND</td>
<td>Linpack Banded</td>
<td>DIA</td>
<td>Diagonal</td>
</tr>
<tr>
<td>COO</td>
<td>Coordinate</td>
<td>BSR</td>
<td>Block Sparse Row</td>
</tr>
<tr>
<td>CSR</td>
<td>Compressed Sparse Row</td>
<td>SSK</td>
<td>Symmetric Skyline</td>
</tr>
<tr>
<td>CSC</td>
<td>Compressed Sparse Column</td>
<td>BSR</td>
<td>Nonsymmetric Skyline</td>
</tr>
<tr>
<td>MSR</td>
<td>Modified CSR</td>
<td>JAD</td>
<td>Jagged Diagonal</td>
</tr>
<tr>
<td>LIL</td>
<td>Linked List</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: CSR = CRS, CCS = CSC, SSK = SKS in some references
\[ A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix} \]

\[ AA = \begin{bmatrix} 3 & 3 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \end{bmatrix} \]

- simple
- row-wise
- easy blocked formats
\[
A = \begin{bmatrix}
1 & 0 & 0 & 2 & 0 \\
3 & 4 & 0 & 5 & 0 \\
6 & 0 & 7 & 8 & 9 \\
0 & 0 & 10 & 11 & 0 \\
0 & 0 & 0 & 0 & 12
\end{bmatrix}
\]

\[
AA = \begin{bmatrix}
12.0 & 9.0 & 7.0 & 5.0 & 1.0 & 2.0 & 11.0 & 3.0 & 6.0 & 4.0 & 8.0 & 10.0
\end{bmatrix}
\]

\[
JR = \begin{bmatrix}
5 & 3 & 3 & 2 & 1 & 1 & 4 & 2 & 3 & 2 & 3 & 4
\end{bmatrix}
\]

\[
JC = \begin{bmatrix}
5 & 5 & 3 & 4 & 1 & 4 & 4 & 1 & 1 & 2 & 4 & 3
\end{bmatrix}
\]

- simple, often used for entry
csr

\[ A = \begin{bmatrix}
1 & 0 & 0 & 2 & 0 \\
3 & 4 & 0 & 5 & 0 \\
6 & 0 & 7 & 8 & 9 \\
0 & 0 & 10 & 11 & 0 \\
0 & 0 & 0 & 0 & 12 \\
\end{bmatrix} \]

\[ AA = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 & 10.0 & 11.0 & 12.0 \\
\end{bmatrix} \]

\[ JA = \begin{bmatrix}
1 & 4 & 1 & 2 & 4 & 1 & 3 & 4 & 5 & 3 & 4 & 5 \\
\end{bmatrix} \]

\[ IA = \begin{bmatrix}
1 & 3 & 6 & 10 & 12 & 13 \\
\end{bmatrix} \]

- Length of AA and JA is \( nnz \); length of IA is \( n + 1 \)
- \( IA(j) \) gives the index (offset) to the beginning of row \( j \) in AA and JA (one origin due to Fortran)
- no structure, fast row access, slow column access
- related: CSC, MSR
\[ A = \begin{bmatrix}
1 & 0 & 0 & 2 & 0 \\
3 & 4 & 0 & 5 & 0 \\
6 & 0 & 7 & 8 & 9 \\
0 & 0 & 10 & 11 & 0 \\
0 & 0 & 0 & 0 & 12
\end{bmatrix} \]

\[ AA = [1.0 \ 4.0 \ 7.0 \ 11.0 \ 12.0 \ 2.0 \ 3.0 \ 5.0 \ 6.0 \ 8.0 \ 9.0 \ 10.0] \]

\[ JA = [7 \ 8 \ 10 \ 13 \ 14 \ 14 \ 4 \ 1 \ 4 \ 1 \ 4 \ 5 \ 3] \]

- places importance on diagonal (often nonzero and accessed frequently)
- first \( n \) entries are the diag
- \( n + 1 \) is empty
- rest of \( AA \) are the nondiagonal entries
- first \( n + 1 \) entries in \( JA \) give the index (offset) of the beginning of the row (the \( IA \) of CSR is in this \( JA \))
- rest of \( JA \) are the columns indices
\[ A = \begin{bmatrix}
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{bmatrix} \quad \text{DIAG} = \begin{bmatrix}
* & 1.0 & 2.0 \\
3.0 & 4.0 & 5.0 \\
6.0 & 7.0 & 8.0 \\
9.0 & 10.0 & * \\
11.0 & 12.0 & *
\end{bmatrix} \quad \text{IOFF} = [-1, 0, 2]

- need to know the offset structure
- some entries will always be empty
try it...

\[ A = \begin{bmatrix}
7 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 0 & 0 & 0 \\
0 & 2 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 0 & 6 & 4
\end{bmatrix} \]

- CSR
- COO
\[
A = \begin{bmatrix}
7 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 0 & 0 & 0 \\
0 & 2 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 0 & 6 & 4 \\
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>IA</th>
<th>JA</th>
<th>AA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\[
\text{COO}
\]

\[
\begin{array}{c|c|c|c|}
\text{i} & \text{IA} & \text{JA} & \text{AA} \\
\hline
1 & 1 & 1 & 7 \\
2 & 2 & 2 & 1 \\
3 & 4 & 3 & 2 \\
4 & 6 & 2 & 2 \\
5 & 7 & 4 & 2 \\
6 & 9 & 5 & 5 \\
7 & - & 5 & 6 \\
8 & - & 6 & 4 \\
\end{array}
\]

\[
\text{CSR}
\]
sparse matrix-vector multiply

\[ z = Ax, \quad A_{m \times n}, \quad x_{n \times 1}, \quad z_{m \times 1} \]

1. **input** \( A, \ x \)
2. \( z = 0 \)
3. **for** \( i = 1 \) to \( m \)
4. \hspace{1em} **for** \( col = A(i,:) \)
5. \hspace{2em} \( z(i) = z(i) + A(i, col) \cdot x(col) \)
6. \hspace{1em} **end**
7. **end**
sparse matrix-vector multiply

\[ z = Ax, \quad A_{m \times n}, \quad x_{n \times 1}, \quad z_{m \times 1} \]

```
1 DO  I=1, m
2     Z(I)=0
3   K1 = IA(I)
4   K2 = IA(I+1)-1
5  DO  J=K1, K2
6      z(I) = z(I) + A(J)*x(JA(J))
7  ENDDO
8 ENDDO
```

- \( O(nnz) \)
- marches down the rows
- very cheap
sparse matrix-matrix multiply

- ways to optimize (“SMPP”, Douglas, Bank)

\[ Z = AB, \ A_{m \times n}, \ B_{n \times p}, \ z_{m \times p} \]

```plaintext
1 for i = 1 to m
2   for j = 1 to n
3       Z(i, j) = dot(A(i,:), B(:,j))
4   end
5 end
6 return Z
```

- obvious problem: column selection of \( B \) is expensive for CSR
- not-so-obvious problem: \( Z \) is sparse(!!), but the algorithm doesn’t account for this.
sparse matrix-matrix multiply

\[ Z = AB, \ A_{m \times n}, \ B_{n \times p}, \ z_{m \times p} \]

```
Z=0
for i = 1 to m
    for colA = A(i,:)
        for colB = A(colA,:)
            Z(i, colB) += A(i, colA) \cdot B(colA, colB)
        end
    end
end
return Z
```

- only marches down rows
- only computes nonzero entries in \( Z \) (aside from fortuitous subtractions)
- line 5 will do and insert into \( Z \). Two options:
  1. precompute sparsity of \( Z \) in CSR
  2. use LIL for \( Z \)
some python

\[
A = \begin{bmatrix}
7 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 0 & 0 & 0 \\
0 & 2 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 0 & 6 & 4 \\
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th></th>
<th>IA</th>
<th>JA</th>
<th>AA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

```python
from scipy import sparse
from numpy import array
IA=array([1,2,3,1,4,0,4,2])
JA=array([1,3,4,2,5,0,4,1])
V=array([1,2,5,2,4,7,6,2])
A=sparse.coo_matrix((V,(IA,JA)),shape=(5,6))
```
some python

From COO to CSC:

```python
from scipy import sparse
from numpy import array
import pprint
IA = array([1, 2, 3, 1, 4, 0, 4, 2])
JA = array([1, 3, 4, 2, 5, 0, 4, 1])
V = array([1, 2, 5, 2, 4, 7, 6, 2])
A = sparse.coo_matrix((V, (IA, JA)), shape=(5, 6)).tocsr()
```

Nonzeros:

```python
print(A.nnz)
```

To full and view:

```python
B = A.todense()
pprint.pprint(B)
```
simple matrix iterations

- Solve
  \[ Ax = b \]
- Assumption: \( A \) is very sparse
- Let \( A = N + M \), then
  \[ Ax = b \]
  \[ (N + M)x = b \]
  \[ Nx = b - Mx \]
- Make this into an iteration:
  \[ Nx_k = b - Mx_{k-1} \]
  \[ x_k = N^{-1}(b - Mx_{k-1}) \]
- Careful choice of \( N \) and \( M \) can give effective methods
- More powerful iterative methods exist