ECEN 615
Methods of Electric Power Systems Analysis

Lecture 9: Sparse Systems

Prof. Tom Overbye
Dept. of Electrical and Computer Engineering
Texas A&M University
overbye@tamu.edu

Announcements

- Read Chapter 7 from the book
- Homework 2 is due on Thursday September 26

ECEN 615 Sparsity Limitations

- Sparse matrices arise in many areas, and can have domain specific structures
  - Symmetric matrices
  - Structurally symmetric matrices
  - Tridiagnonal matrices
  - Banded matrices
- ECEN 615 is focused on problems that arise in the electric power; it is not a general sparse matrix course

Full Matrix versus Sparse Matrix Storage

- Full matrices are easily stored in arrays with just one variable needed to store each value since the value's row and column are implicitly available from its matrix position
- With sparse matrices two or three elements are needed to store each value
  - The zero values are not explicitly stored
  - The value itself, its row number and its column number
  - Storage can be reduced by storing all the elements in a particular row or column together
- Because large matrices are often quite sparse, the total storage is still substantially reduced

Sparse Matrix Usage Can Determine the Optimal Storage

- How a sparse matrix is used can determine the best storage scheme to use
  - Row versus column access; does structure change
- Is the matrix essentially used only once? That is, its structure and values are assumed new each time used
- Is the matrix structure constant, with its values changed
  - This would be common in the N-R power flow, in which the structure doesn’t change each iteration, but its values do
- Is the matrix structure and values constant, with just the **b** vector in **Ax=b** changing
  - Quite common in transient stability solutions

Numerical Precision

- Required numerical precision determines type of variables used to represent numbers
  - Specified as number of bytes, and whether signed or not
- For Integers
  - One byte is either 0 to 255 or -128 to 127
  - Two bytes is either smallint (-32,768 to 32,767) or word (0 to 65,536)
  - Four bytes is either Integer (-2,147,483,648 to 2,147,483,647) or Cardinal (0 to 4,294,967,295)
    - This is usually sufficient for power system row/column numbers
  - Eight bytes (Int64) if four bytes is not enough

Numerical Precision, cont.

- For floating point values using choice is between four bytes (single precision) or eight bytes (double precision); extended precision has ten bytes
  - Single precision allows for 6 to 7 significant digits
  - Double precision allows for 15 to 17 significant digits
  - Extended allows for about 18 significant digits
  - More bytes requires more storage
  - Computational impacts depend on the underlying device; on PCs there isn't much impact; GPUs can be 3 to 8 times slower for double precision
- For most power problems double precision is best

General Sparse Matrix Storage

- A general approach for storing a sparse matrix would be using three vectors, each dimensioned to number of elements
  - AA: Stores the values, usually in power system analysis as double precision values (8 bytes)
  - JR: Stores the row number; for power problems usually as an integer (4 bytes)
  - JC: Stores the column number, again as an integer
- If unsorted then both row and column are needed
- New elements could easily be added, but costly to delete
- Unordered approach doesn't make for good computation since elements used next computationally aren't necessarily nearby
- Usually ordered, either by row or column

Sparse Storage Example

- Assume
$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$

- Then
$$\mathbf{AA} = \begin{bmatrix} 5 & -4 & 4 & -3 & 3 & -2 & -4 & -3 & -2 & 10 \end{bmatrix}$$
$$\mathbf{JR} = \begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4 & 4 \end{bmatrix}$$
$$\mathbf{JC} = \begin{bmatrix} 1 & 4 & 2 & 4 & 3 & 4 & 1 & 2 & 3 & 4 \end{bmatrix}$$

Note, this example is a symmetric matrix, but the technique is general

Compressed Sparse Row Storage

- If elements are ordered (as was case for previous example) storage can be further reduced by noting we do not need to continually store each row number
- A common method for storing sparse matrices is known as the Compressed Sparse Row (CSR) format
  - Values are stored row by row
  - Has three vector arrays:
    - AA: Stores the values as before
    - JA: Stores the column index (done by JC in previous example)
    - IA: Stores the pointer to the index of the beginning of each row

CSR Format Example

- Assume as before

$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$

- Then

$$\mathbf{AA} = \begin{bmatrix} 5 & -4 & 4 & -3 & 3 & -2 & -4 & -3 & -2 & 10 \end{bmatrix}$$

$$\mathbf{JA} = \begin{bmatrix} 1 & 4 & 2 & 4 & 3 & 4 & 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\mathbf{IA} = \begin{bmatrix} 1 & 3 & 5 & 7 \end{bmatrix}$$

CSR Comments

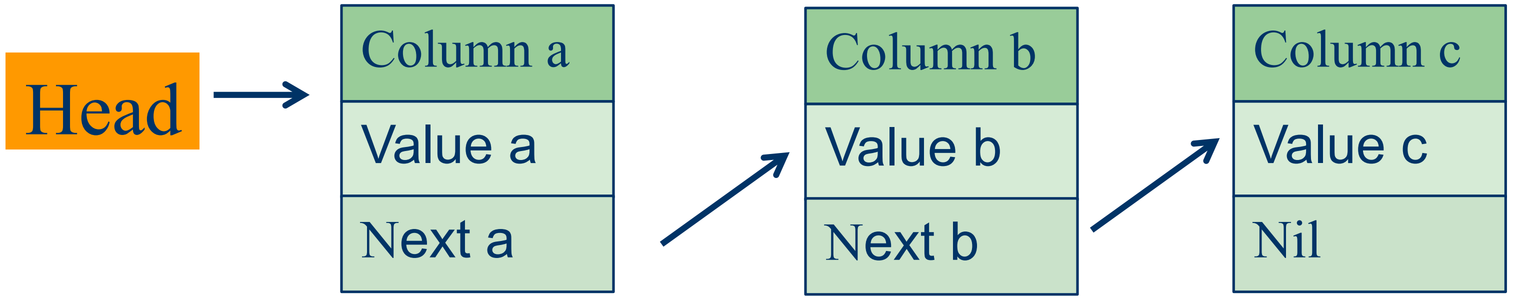
- The CSR format reduces the storage requirements by taking advantage of needing only one element per row
- The CSR format has good advantages for computation when using cache since (as we shall see) during matrix operations we are often sequentially going through the vectors
- An alternative approach is Compressed Sparse Column (CSC), which identical, except storing the values by column
- It is difficult to add values.
- We'll mostly use the linked list approach here, which makes matrix manipulation simpler

Linked Lists: Classes and Objects

- In explaining the linked list approach it is helpful to use the concepts from object oriented programming (OOP) of classes and objects
  - Approach can also be used in non-OOP programming
- OOP can be thought of as a collection of objects interacting with each other
- Objects are instances of classes.
- Classes define the object fields and actions (methods)
- We'll define a class called sparse matrix element, with fields of value, column and next; each sparse matrix element is then an object of this class

Linked Lists

- A linked list is just a group of objects that represent a sequence
  - We'll used linked lists to represent a row or column of a sparse matrix
- Each linked list has a head pointer that points to the first object in the list
  - For our sparse matrices the head pointer will be a vector of the rows or columns



Sparse Matrix Storage with Linked Lists by Rows

- If we have an n by n matrix, setup a class called TSparseElement with fields column, value and next
- Setup an n-dimensional head pointer vector that points to the first element in each row
- Each nonzero corresponds to an object of class (type) TSparseElement
- We do not need to store the row number since it is given by the object's row
- For power system sparse matrices, which have nonzero diagonals, we also have a header pointer vector that points to the diagonal objects

Linked Lists, cont.

- Linked lists can be singly linked, which means they just go in one direction (to the next element), or doubly linked, pointing to both the previous and next elements
  - Mostly we'll just need singularly linked lists
- With linked lists it is quite easy to add new elements to the list. This can be done in sorted order just by going down the list until the desired point is reached, then changing the next pointer for the previous element to the new element, and for the new element to the next element (for a singly linked list)

On Board Example

- Draw the data structures for the matrix

$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$

Example Pascal Code for Writing a Sparse Matrix

```
Procedure TSparMat.SMWriteMatlab(Var ft : Text; variableName : String; digits,rod : Integer;
                                 ignoreZero : Boolean; local_MinValue : Double);
Var j : Integer;
  p1 : TMatEle;
Begin
For j := 1 to n Do Begin
  p1 := Row(j).Head;
  While p1 <> nil Do Begin
    If (not IgnoreZero) or (abs(p1.value) > local_MinValue) Then Begin
      If variableName <> '' Then Writeln(ft,variableName+'(',(j),',',(p1.col),')=',p1.value:digits:rod,';')
      Else Writeln(ft,j:5,' ',p1.col:5,' ',p1.value:digits:rod);
    End;
    p1 := p1.next;
  End;
 End;
End;
```

Sparse Working Row

- Before showing a sparse LU factorization it is useful to introduce the concept of a working row full vector
- This is useful because sometimes we need direct access to a particular value in a row
- The working row approach is to define a vector of dimension n and set all the values to zero
- We can then load a sparse row into the vector, with computation equal to the number of elements in the row
- We can then unload the sparse row from the vector by storing the new values in the linked list, and resetting the vector values we changed to zero

Loading the Sparse Working Row

```
Procedure TSparMat.LoadSWRbyCol(rowJ : Integer; var SWR : PDVectorList);
Var p1 : TMatEle;
Begin
  p1 := rowHead[rowJ];
  While p1 <> nil Do Begin
    SWR[p1.col] := p1.value;
    p1 := p1.next;
  End;
End;
```

Unloading the Sparse Working Row

```
Procedure TSParMat.UnLoadSWRbyCol(rowJ : Integer; var SWR :
PDVectorList);
Var p1 : TMatEle;
Begin
  p1 := rowHead[rowJ];
  While p1 <> nil Do Begin
    p1.value := SWR[p1.col];
    SWR[p1.col] := 0;
    p1 := p1.next;
  End;
End;
```

Note, there is no need to explicitly zero out all the elements each iteration since 1) most are still zero and 2) doing so would make it O(n^2). The above code efficiently zeros out just the values that have changed.

Doing an LU Factorization of a Sparse Matrix with Linked Lists

- Now we can show how to do an LU factorization of a sparse matrix stored using linked lists
- We will assume the head pointers are in the vector RowHead, and the diagonals in RowDiag
- Recall this was the approach for the full matrix

```
For i := 2 to n Do Begin  // This is the row being processed
  For j := 1 to i-1 Do Begin  // Rows subtracted from row i
    A[i,j] = A[i,j]/A[j,j]  // This is the scaling
    For k := j+1 to n Do Begin  // Go through each column in i
      A[i,k] = A[i,k] - A[i,j]*A[j,k]
    End;
  End;
End;
```

Sparse Factorization

- Note, if you know about fills, we will get to that shortly; if you don't know don't worry about it yet
- We'll just be dealing with structurally symmetric matrices (incidence-symmetric)
- We'll assume the row linked lists are ordered by column; we'll show how this can be done quickly later
- We will again sequentially going through the rows, starting with row 2, going to row n

```
For i := 2 to n Do Begin  // This is the row being processed
```

Sparse Factorization, cont.

- The next step is to go down row i, up to but not including the diagonal element
- We'll be modifying the elements in row i, so we need to load them into the working row vector
- Key sparsity insight is in doing the below code we only need to consider the non-zeros in A[i,j]; for a full matrix the code is

```
For j := 1 to i-1 Do Begin  // Rows subtracted from row
    A[i,j] = A[i,j]/A[j,j]  // This is the scaling
    For k := j+1 to n Do Begin  // Go through each column in i
            A[i,k] = A[i,k] - A[i,j]*A[j,k]
    End;
```

Sparse Factorization, cont.

```
For i := 1 to n Do Begin   // Start at 1, but nothing to do in first row
 LoadSWRbyCol(i,SWR);   // Load Sparse Working Row }
 p2 := rowHead[i]
 While p2 <> rowDiag[i] Do Begin    // This is doing the j loop
   p1 := rowDiag[p2.col];
   SWR[p2.col] := SWR[p2.col] / p1.value;
   p1 := p1.next;
   While p1 <> nil Do Begin   // Go to the end of the row
     SWR[p1.col] := SWR[p1.col] - SWR[p2.col] *p1.value;
     p1 := p1.next;
   End;
   p2 := p2.next;
  End;
  UnloadSWRByCol(i,SWR);
End;
```

Sparse Factorization Example

- Believe it or not, that is all there is to it! The factorization code itself is quite simple.
- However, there are a few issues we'll get to in a second. But first an example

$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$

- Notice with this example there is nothing to do with rows 1, 2 and 3 since there is nothing before the diag (p2 will be equal to the diag for the first three rows)

Sparse Factorization Example, Cont.

- Doing factorization with i=4
  - Row 4 is full so initially p2= A[4,1] // column 1
  - SWR = [-4 -3 -2 10]
  - p1= A[1,1]
  - SWR[1] = -4/A[1,1] = -4/5 = -0.8
  - p1 goes to A[1,4]

That is, the next element in row 1

  - SWR[4] = 10 – SWR[p2.col]*p1.value = 10 – (-0.8)*-4=6.8
  - p1 = nil; go to next col
  - p2 =A[4,2] // column 2
  - P1 = A[2,2]
  - SWR[2] = -3/A[2,2]= -3/4 = -0.75

Sparse Factorization Example, Cont.

- p1 goes to A[2,4] // p2=A[4,2] The next element in row 2
- SWR[4] = 6.8 – SWR[p2.col]*p1.value = 6.8 – (-0.75)*-3=4.55
- p1 = nil; go to next col
- p2 =A[4,3] // column 3
- p1 = A[3,3]
- SWR[3] = -/A[2,2]= -2/3 = -0.667
- p1 goes to A[3,4] // p2 = A[4,3] The next element in row 3
- SWR[4] = 4.55 – SWR[p2.col]*p1.value
  = 4.55 – (-0.667)*-2=3.2167
- Unload the SWR = [-0.8 -0.75 -0.667 3.2167]
- p2 = A[4,4] = diag so done

Sparse Factorization Examples, Cont.

$$\mathbf{A}_{\text{Factored}} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -0.8 & -0.75 & -0.6667 & 3.2167 \end{bmatrix}$$

- For a second example, again consider the same system, except with the nodes renumbered

$$\mathbf{B} = \begin{bmatrix} 10 & -4 & -3 & -2 \\ -4 & 5 & 0 & 0 \\ -3 & 0 & 4 & 0 \\ -2 & 0 & 0 & 3 \end{bmatrix}$$

Sparse Factorization Examples, Cont.

- With i=2, load SWR = [-4 5 0 0]
  - p2 = B[2,1]
  - p1 = B[1,1]
  - SWR[1]=-4/p1.value=-4/10 = -0.4
  - p1 = B[1,2]
  - SWR[2]=5 – (-0.4)*(-4) = 1.6
  - p1 = B[1,3]
  - SWR[3]= 0 – (-0.4)*(-3) = -1.2
  - p1 = B[1,4]
  - SWR[4]=0 – (-0.4)*(-2) = -0.8
  - p2=p2.next=diag so done
  - UnloadSWR and **we have a problem!**

$$\mathbf{B} = \begin{bmatrix} 10 & -4 & -3 & -2 \\ -4 & 5 & 0 & 0 \\ -3 & 0 & 4 & 0 \\ -2 & 0 & 0 & 3 \end{bmatrix}$$

There are no elements in row 2 for columns 3 and 4!

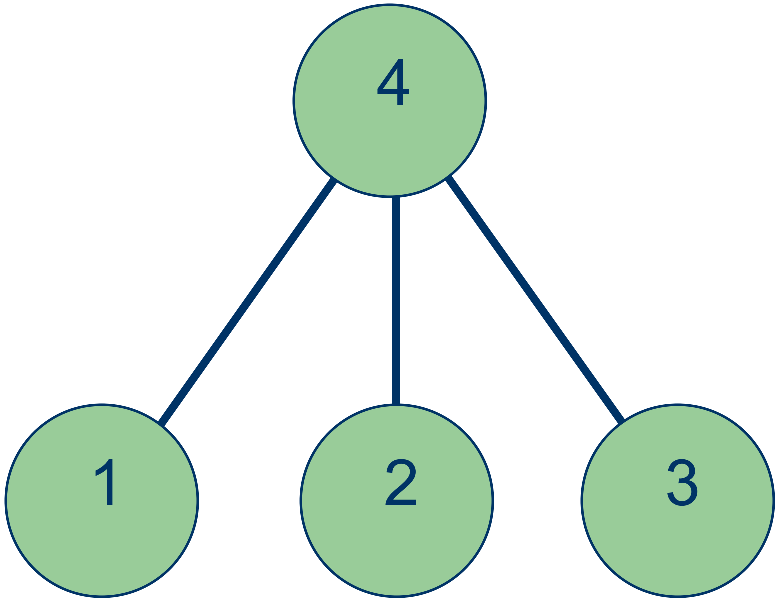
Fills

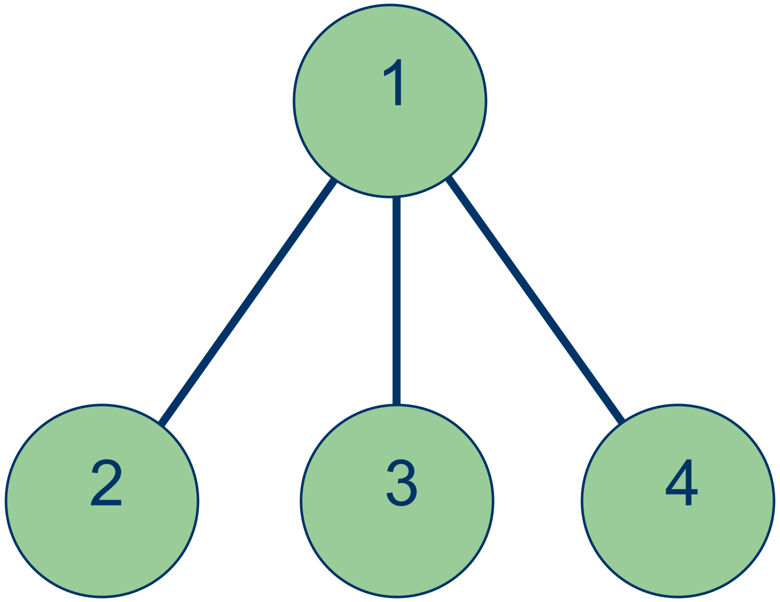
- When doing a factorization of a sparse matrix some values that were originally zero can become nonzero during the factorization process
- These new values are called “fills” (some call them fill-ins)
- For a structurally symmetric matrix the fill occurs for both the element and its transpose value (i.e., $\mathbf{A}_{ij}$ and $\mathbf{A}_{ji}$)
- How many fills are required depends on how the matrix is ordered
  - For a power system case this depends on the bus ordering

Fills

- There are two key issues associated with fills
  - Adding the fills
  - Ordering the matrix elements (buses in our case) to reduce the number of fills
- The amount of computation required to factor a sparse matrix depends upon the number of nonzeros in the original matrix, and the number of fills added
- How the matrix is ordered can have a dramatic impact on the number of fills, and hence the required computation
- Usually a matrix cannot be ordered to totally eliminate fills

Fill Examples





$$\mathbf{A} = \begin{bmatrix} 5 & 0 & 0 & -4 \\ 0 & 4 & 0 & -3 \\ 0 & 0 & 3 & -2 \\ -4 & -3 & -2 & 10 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 10 & -4 & -3 & -2 \\ -4 & 5 & 0 & 0 \\ -3 & 0 & 4 & 0 \\ -2 & 0 & 0 & 3 \end{bmatrix}$$

No Fills Required

Fills Required (matrix becomes full)

Example: 7 by 7 Matrix

- Consider the 7 x 7 matrix **A** with the zero-nonzero pattern shown in (*a*): of the 49 possible elements there are only 31 that are nonzero
- If elimination proceeds with the given ordering, all but two of the 18 originally zero entries, will fill in, as seen in (*b*)

Example: 7 by 7 Matrix Structure

r \ c	1	2	3	4	5	6	7
1	X	X	X	X	X	X	
2	X	X	X			X	X
3	X	X	X			X	X
4	X			X	X		
5	X			X	X	X	
6	X	X	X		X	X	
7		X	X				X

(a) The original zero-nonzero structure

r \ c	1	2	3	4	5	6	7
1	X	X	X	X	X	X	
2	X	X	X	F	F	X	X
3	X	X	X	F	F	X	X
4	X	F	F	X	X	F	F
5	X	F	F	X	X	X	F
6	X	X	X	F	X	X	F
7		X	X	F	F	F	X

(b) The post- elimination zero nonzero pattern

Example: 7 by 7 Matrix Reordering

- We next reorder the rows and the columns of **A** so as to result in the pattern shown in (c)
- For this reordering, we obtain no fills, as shown in the table of factors given in (d)
- In this way, we preserve the original sparsity of **A**

Example: 7 by 7 Matrix Reordered Structure

r \ *c*	4	5	1	6	7	3	2
4	*X*	*X*	*X*				
5	*X*	*X*	*X*	*X*			
1	*X*	*X*	*X*	*X*		*X*	*X*
6		*X*	*X*	*X*		*X*	*X*
7					*X*	*X*	*X*
3			*X*	*X*	*X*	*X*	*X*
2			*X*	*X*	*X*	*X*	*X*

(c) The reordered system

r \ *c*	4	5	1	6	7	3	2
4	*X*	*X*	*X*				
5	*X*	*X*	*X*	*X*			
1	*X*	*X*	*X*	*X*		*X*	*X*
6		*X*	*X*	*X*		*X*	*X*
7					*X*	*X*	*X*
3			*X*	*X*	*X*	*X*	*X*
2			*X*	*X*	*X*	*X*	*X*

(d) The post- elimination reordered system

Fills for Structurally Symmetric Matrices

- For structurally symmetric matrices we can gain good insights into the problem by examining the graph-theoretic interpretation of the triangularization process
- This assumption involves essentially no loss in generality since if $\mathbf{A}_{ij} \neq 0$ but $\mathbf{A}_{ji} = 0$ we simply treat $\mathbf{A}_{ji}$ as a nonzero element with the value 0; in this way, we ensure that $\mathbf{A}$ has a symmetric structure
- We term a matrix as structurally symmetric whenever it has a symmetric zero-nonzero pattern

Graph Associated with A

- We make use of graph theoretic notions to develop a practical reordering scheme for sparse systems
- We associate a graph G with the zero-nonzero structure of the *n by n* matrix **A**
- We construct the graph G associated with the matrix **A** as follows:

  i. G has *n* nodes corresponding to the dimension *n* of the square matrix: node *i* represents both the column *i* and the row *i* of **A**;

  ii. a branch (*k, j*) connecting nodes *k* and *j* exists if and only if the element $\mathbf{A}_{jk}$ (and, by structural symmetry, $\mathbf{A}_{kj}$) is nonzero; the self loop corresponding to $\mathbf{A}_{kk}$ is not represented

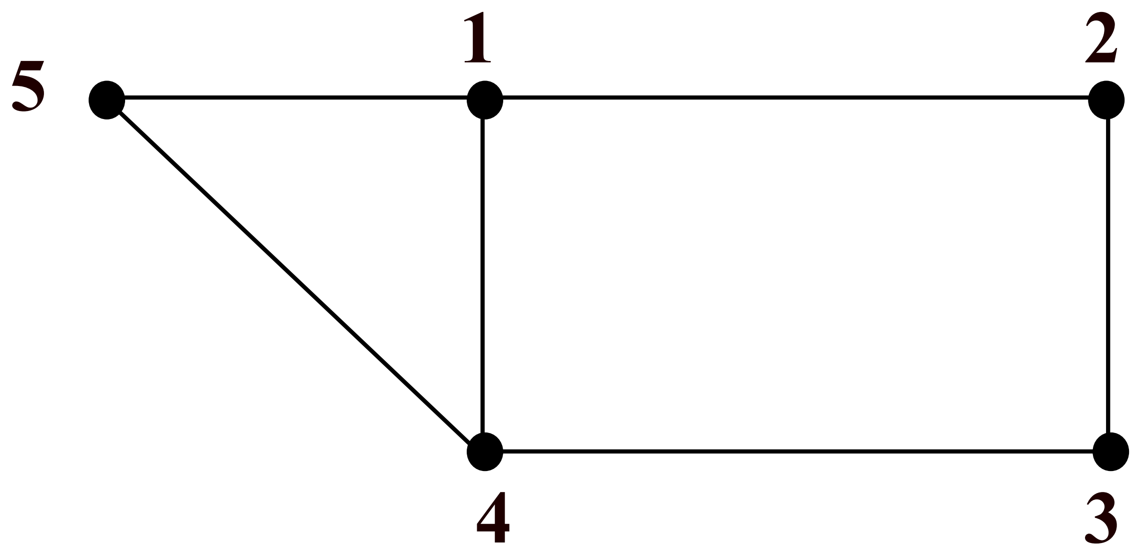
Example: 5 by 5 System

- Suppose that **A** has the zero-nonzero pattern

r \ *c*	1	2	3	4	5
1	*X*	*X*		*X*	*X*
2	*X*	*X*	*X*		
3		*X*	*X*	*X*	
4	*X*		*X*	*X*	*X*
5	*X*			*X*	*X*

Example: 5 by 5 System

- Then, the associated graph G is



Graph-Theoretic Interpretation

- The graph-theoretic interpretation of the elimination of the node (bus) j is as follows
- The deletion of the node j involves all its incident branches (k, j) and (j, k) connected to j, k≠j
- In the pre-elimination graph of the eliminated node j, the elimination of the branches (j, k) and (l, j) results in the addition of the new branch (k, l), if one does not already exist

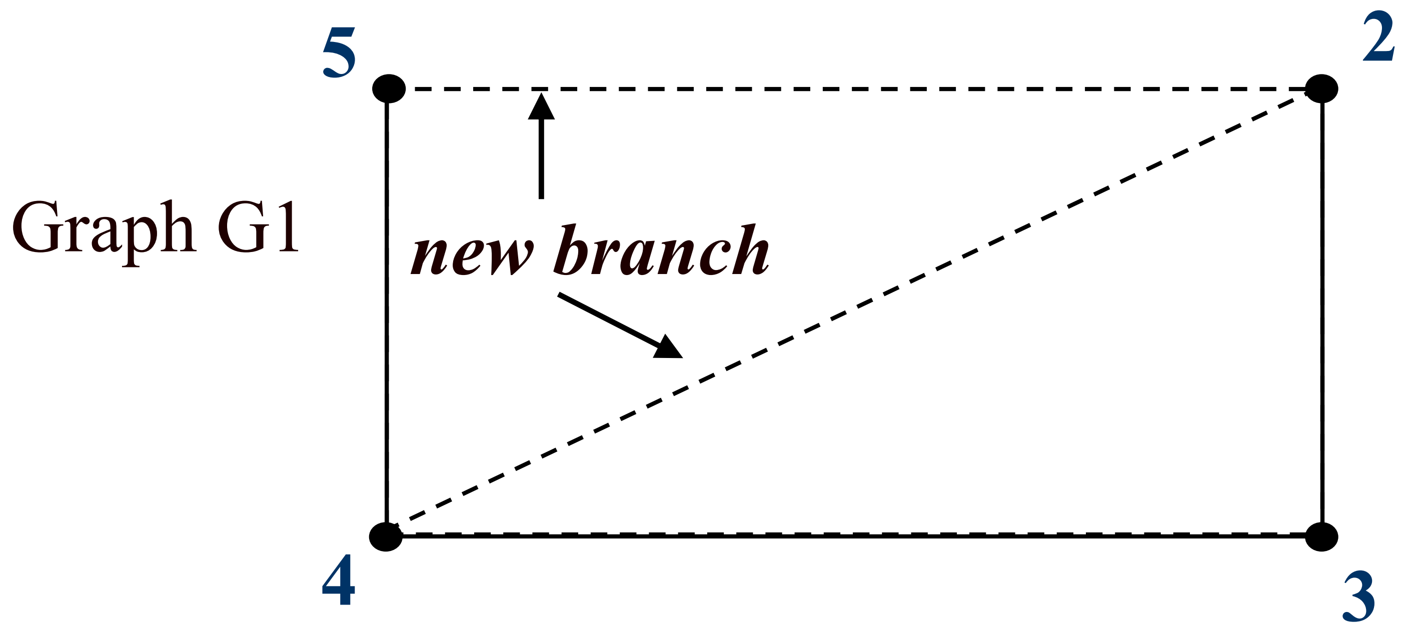
Example: 5 by 5 System

- We eliminate the Bus (Node) 1 variable with the resulting zero-nonzero pattern as shown the array

r \ c	1	2	3	4	5
1	X	X		X	X
2	X	X	X	F	F
3		X	X	X	
4	X	F	X	X	X
5	X	F		X	X

Bordered by the broken lines with on the new graph new G1

Example: 5 by 5 System



- We obtain the graph G1 from G by removing Bus 1 with the new added branches (2, 4) and (2, 5) corresponding to the fills

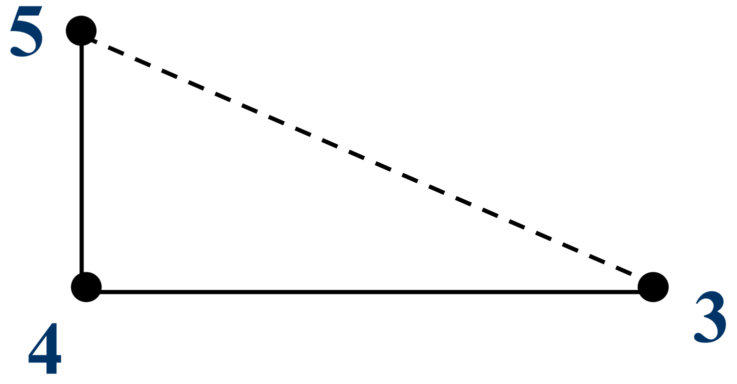
Example: 5 by 5 System

- The elimination of Bus 2 results in the submatrix shown below

r \ *c*	3	4	5
3	*X*	*X*	*F*
4	*X*	*X*	*X*
5	*F*	*X*	*X*

Example: 5 by 5 System

with the corresponding graph G2

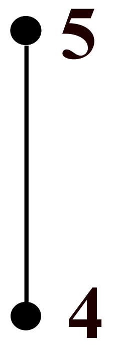


- The elimination of Bus 3 yields

r \ c	4	5
4	*X*	*X*
5	*X*	*X*

Example: 5 by 5 System

with the corresponding graph G3



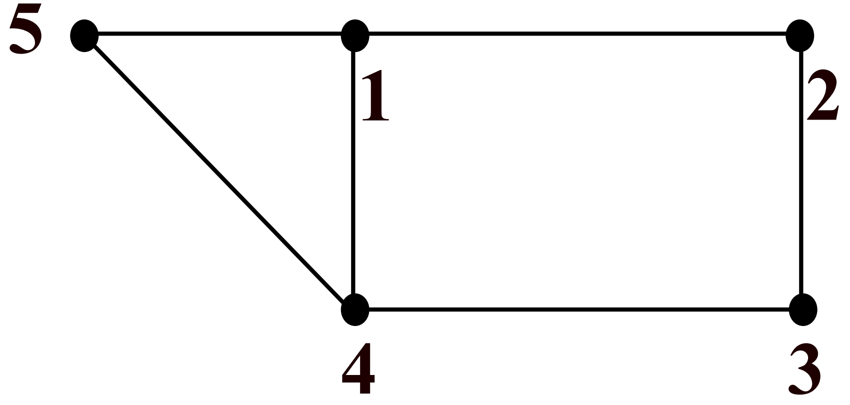
- Finally, upon Bus 4 we have

$$\begin{array}{c|c} {}_{r}\diagdown^{c} & 5 \\ \hline 5 & X \end{array}$$

- and the corresponding G4 is simply the point

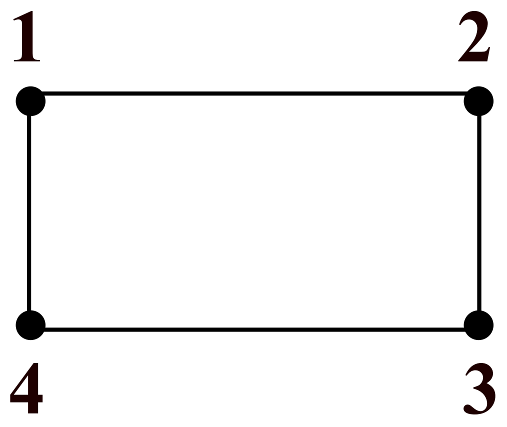
Reording the Rows/Columns

- We next examine how we may reorder the rows and columns of **A** to preserve its sparsity, i.e., to minimize the number of fills
- Eventually we'll introduce an algorithm to try to minimize the fills
- This is motivated by revisiting the graph G



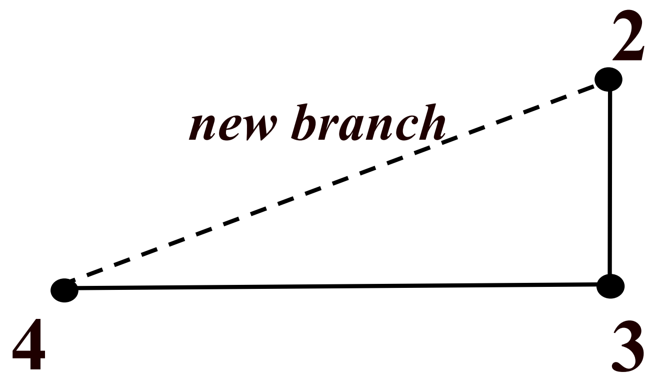
Reording Motivating Example

- To minimize the number of fills, i.e., the number of new branches in G, we eliminate first the node which upon deletion introduces the least number of new branches
- This is node 5 and upon deletion no new branches are added and the resulting graph G1 is



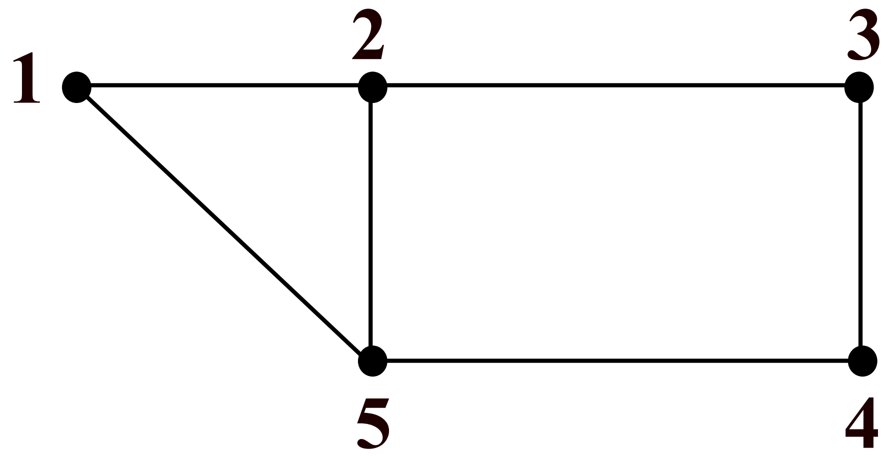
Reording Motivating Example

- The structure of G1 is such that any one of the remaining nodes may be chosen as the next node to be eliminated since each of the 4 remaining nodes introduces a new branch after its elimination
- We arbitrarily pick node 1 and we obtain the graph G2
- We continue with the next three choices arbitrary, resulting in no new fills



Reording Motivating Example

- We may relabel the original graph in such a way that the label of the node refers to the order in which it is eliminated
- Thus we renumber the nodes as shown below



Reording Motivating Example

- Clearly, relabeling the nodes corresponds to reordering the rows and columns of **A**
- For the reordered system, the zero-nonzero pattern of **A** is

r \ *c*	1	2	3	4	5
1	*X*	*X*			*X*
2	*X*	*X*	*X*		*X*
3		*X*	*X*	*X*	
4			*X*	*X*	*X*
5	*X*	*X*		*X*	*X*

Reording Motivating Example

and of its table of factors has the zero-nonzero structure

r \ c	1	2	3	4	5
1	*X*	*X*			*X*
2	*X*	*X*	*X*		*X*
3		*X*	*X*	*X*	*F*
4			*X*	*X*	*X*
5	*X*	*X*	*F*	*X*	*X*

Compared to the original ordering scheme, the new ordering scheme has saved us 4 fill-ins

General Findings

- The associated graph of the structurally symmetric matrix **A** is useful in gaining insights into the factorization process
- We make the following observations
  - If **A** is originally structurally symmetric, then it remains so in all the steps of the factorization;
  - A good ordering scheme is independent of the values of the elements of **A** and depends only on its the zero-nonzero pattern

Permutation Vectors

- Often the matrix itself is not physically reorded when it is renumbered. Rather we can make use of what is known as a permutation vector, and (if needed) an inverse permutation vector
- These vectors implement the following functions
  - $i_{new} = New(i_{old})$
  - $i_{old} = Old(i_{new})$
- For an n by n matrix the permutation vector is an n-sized integer vector
- If ordered lists are needed, then the linked lists do need to be reordered, but this can be done quickly

Permutation Vectors, cont.

- For the previous five bus example, in which the buses are to be reordered to (5,1,2,3,4), the permutation vector would be **rowPerm**=[5,1,2,3,4]
  - That is, the first row to consider is row 5, then row 1, …
- If needed, the inverse permutation vector is **invRowPerm** = [2,3,4,5,1]
  - That is, with the reordering the first element is in position 2, the second element in position 2, ….
- Hence i = invRowPerm[rowPerm[i]]

Sparse Factorization using a Permutation Vector

```
For i := 1 to n Do Begin
 k = rowPerm[i];  // this is the only change, except using k
 LoadSWRbyCol(k,SWR);   // Load Sparse Working Row }
 p2 := rowHead[k];  // the row needs to be ordered correctly!
 While p2 <> rowDiag[k] Do Begin
   p1 := rowDiag[p2.col];
   SWR[p2.col] := SWR[p2.col] / p1.value;
   p1 := p1.next;
   While p1 <> nil Do Begin   // Go to the end of the row
     SWR[p1.col] := SWR[p1.col] - SWR[p2.col] *p1.value;
     p1 := p1.next;
   End;
   p2 := p2.next;
 End;
 UnloadSWRByCol(k,SWR);
End;
```

Sparse Matrix Reordering

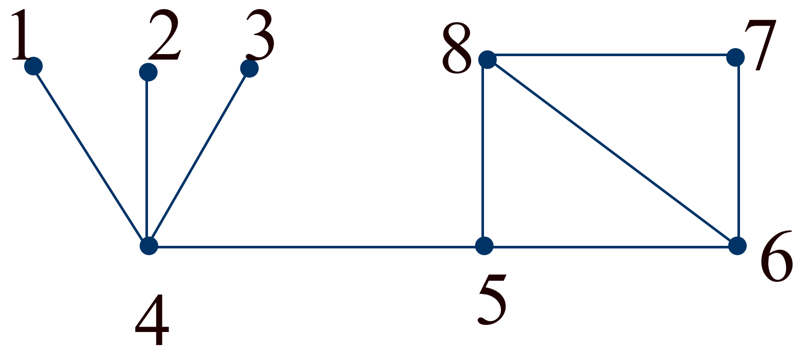
- There is no computationally efficient way to optimally reorder a sparse matrix; however there are very efficient algorithms to greatly reduce the fills
- Two steps here: 1) order the matrix, 2) add fills
- A quite common algorithm combines ordering the matrix with adding the fills
- The two methods discussed here were presented in the 1963 paper by Sato and Tinney from BPA; known as Tinney Scheme 1 and Tinney Scheme 2 since they are more explicitly described in Tinney's 1967 paper
  - 1967 paper also has Tinney Scheme 3 (briefly covered)

Tinney Scheme 1

- Easy to describe, but not really used since the number of fills, while reduced, is still quite high
- In graph theory the degree (or valence or valency) of a vertex is the number of edges incident to the vertex
- Order the nodes (buses) by the number of incident branches (i.e., its valence) those with the lowest valence are ordered first
  - Nodes with just one incident line result in no new fills
  - Obviously in a large system many nodes will have the same number of incident branches; ties can be handled arbitrarily

Tinney Scheme 1, Cont.

- Once the nodes are reordered, the fills are added
  - Common approach to ties is to take the lower numbered node first
- A shortcoming of this method is as the fills are added the valence of the adjacent nodes changes



Node	Valence
1	1
2	1
3	1
4	4
5	3
6	3
7	2
8	3

Tinney 1 order is 1,2,3,7,5,6,8,4

Number of new branches is 2 (4-8, 4-6)