ECEN 615
Methods of Electric Power Systems Analysis

Lecture 10: Sparsity

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

TEXAS A&M UNIVERSITY

Announcements

- Read Chapter 7.
- Homework 2 is now due on Tuesday Sept 30 (extended during class today from Sept 25).
- Sparse matrices are not covered in the recommended course text; the course slides and lectures should be sufficient.
  - If you want more information there are lots of books available; if you'd like a free book, Prof. Yousef Saad of University of Minnesota provides a pdf for the 2003 edition of his book at www-users.cse.umn.edu/~saad/
- The first exam is on Thursday Oct 9 during class (Sanjana will make arrangements for the distance education students).
  - I posted my first exam from 2022 to Canvas as an example (in the Home and Exams section); I'll post the answers in a week or so

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 go 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 there is 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 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!

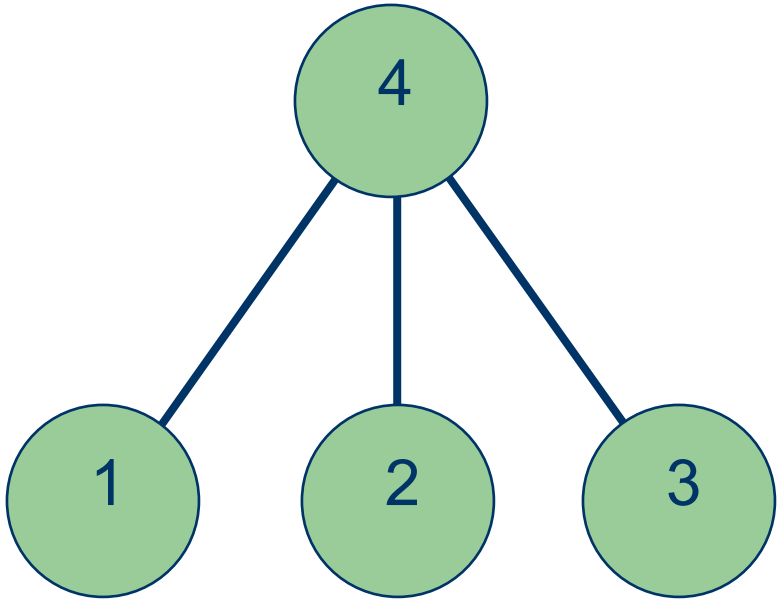
Fills

- 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

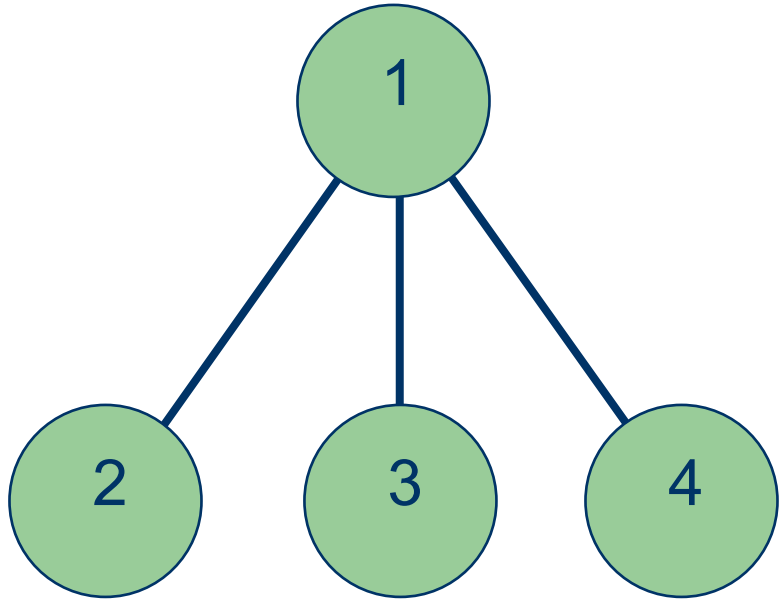
- There are two key issues associated with fills:
  - Adding the fills
  - Ordering the matrix elements (buses in our case) to reduce the number of fills
- The amount of computation required to factor a sparse matrix depends upon the number of nonzeros in the original matrix, and the number of fills 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



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

No Fills Required



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

Fills Required (matrix becomes full)

Example: 7 by 7 Matrix

- Consider the 7 x 7 matrix **A** with the zero-nonzero pattern shown below left; 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 below right.

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			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

The post- elimination zero nonzero pattern

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

Example: 7 by 7 Matrix Reordering

- We next reorder the rows and the columns of **A** so as to result in the pattern shown below left; reordering is just flipping the row order in **A** (and **b**).
- For this reordering, we obtain no fills, as shown below right; in this way, we preserve the original sparsity of **A**

r	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

r	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

Graph (Grid) Insights

- For electric grids it can be helpful to relate the sparse matrix back to an associated electric grid.
- Assume an n by n matrix **A** in which all the diagonals are non-zeros.
- If there is a connection between two buses (nodes), say at position j and k, then the associated matrix entries, $\mathbf{A}_{jk}$ and $\mathbf{A}_{kj}$ are nonzero.

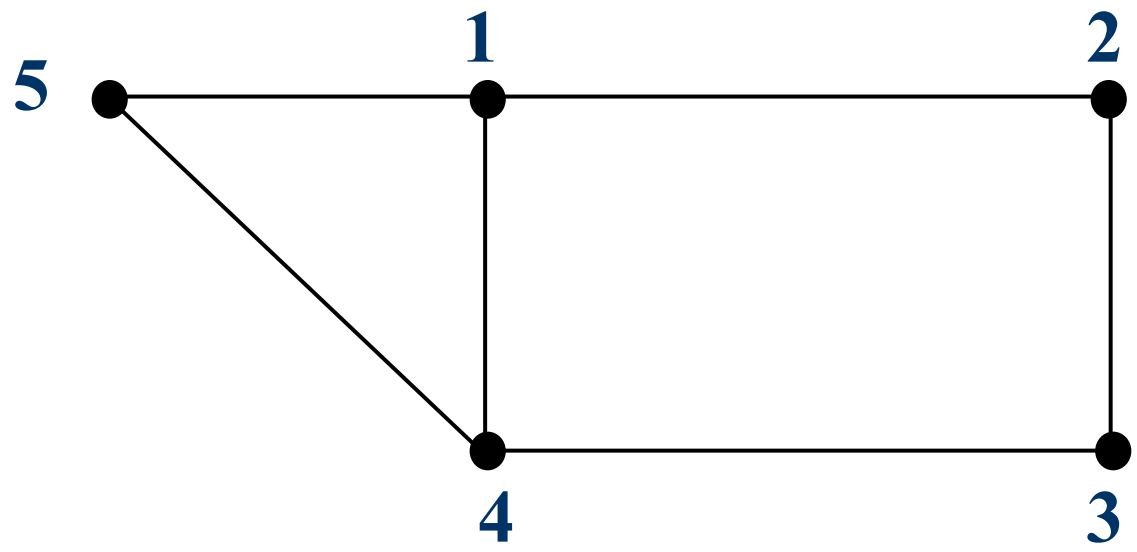
Example: 5 by 5 System

- 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: Associated Graph

- Then, the associated graph G is

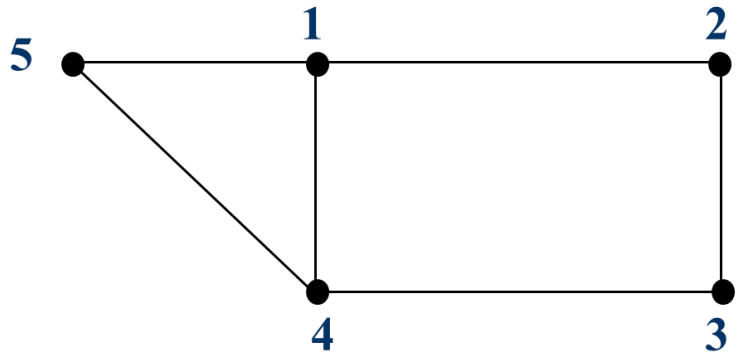


We could also go from the graph to the matrix

Example: 5 by 5 System: Adding Fills for Bus 1

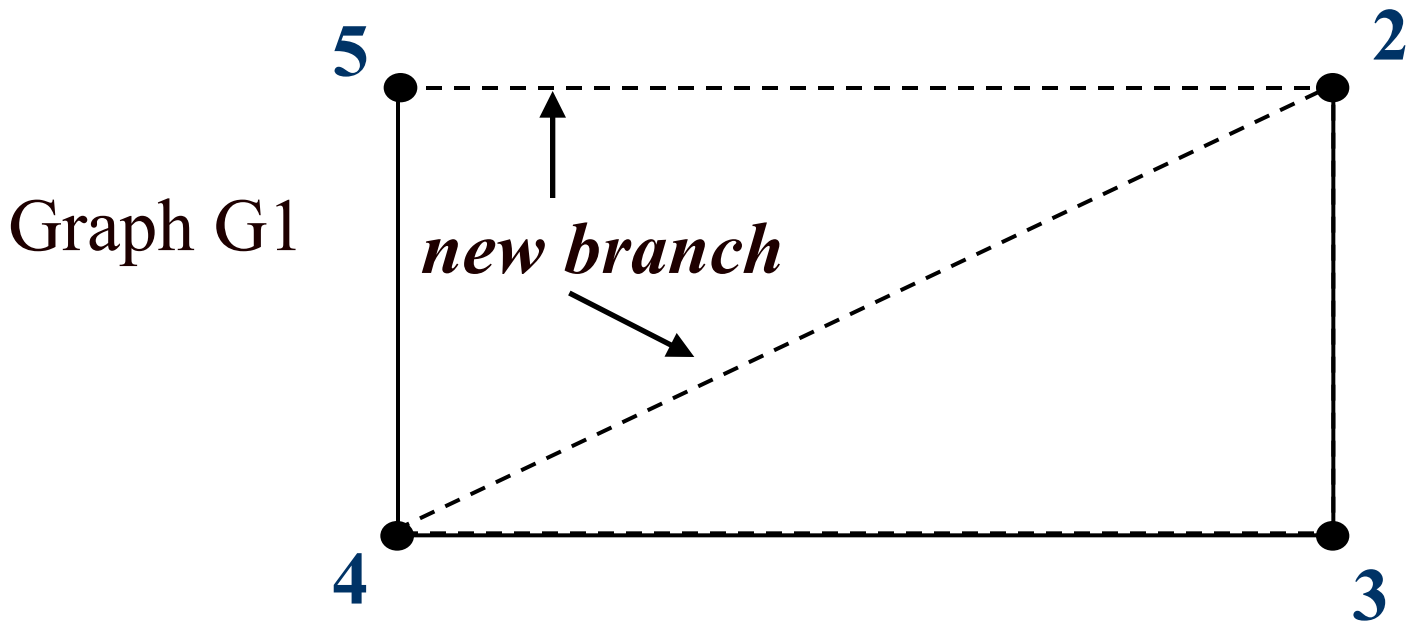
- If we decide to order Bus 1, then it is deleted from the graph, with connections added between its incident buses (2, 4, 5).

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



Here new lines are added between 2 and 4, and between 2 and 5

Example: 5 by 5 System: New Graph



- 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.

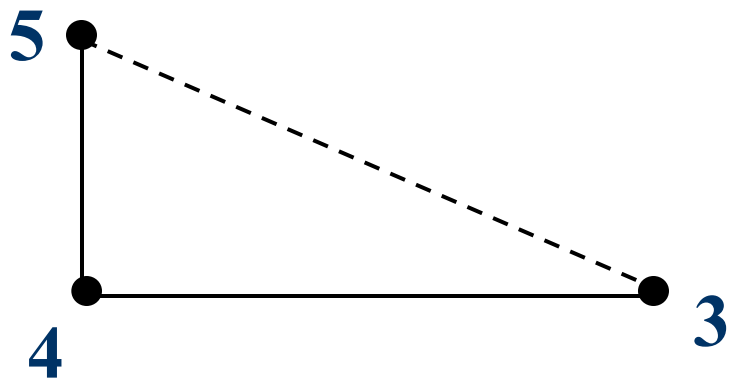
Example: 5 by 5 System: Adding Fills for Bus 2

- 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

Now a new branch is added between buses 3 and 5

- With the corresponding graph



Example: 5 by 5 System: Adding Fills for Bus 3

- The elimination of Bus 3 yields

r \ c	4	5
4	X	X
5	X	X

- with the corresponding graph G3

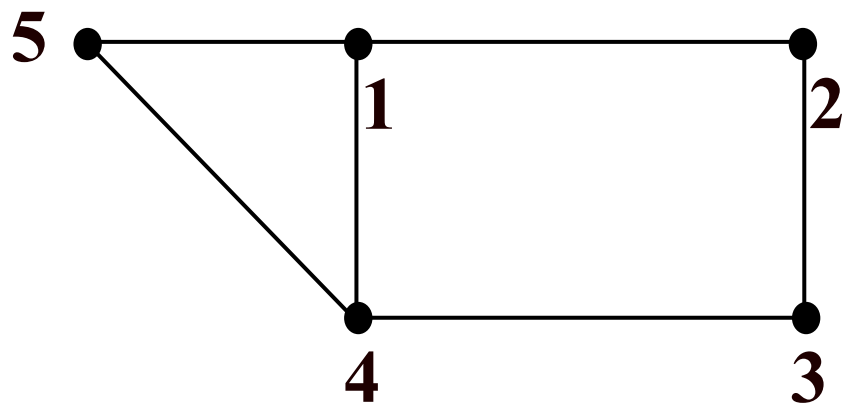
- Finally, upon Bus 4 we have

r \ c	5
5	X

and the corresponding G4 is simply the point.

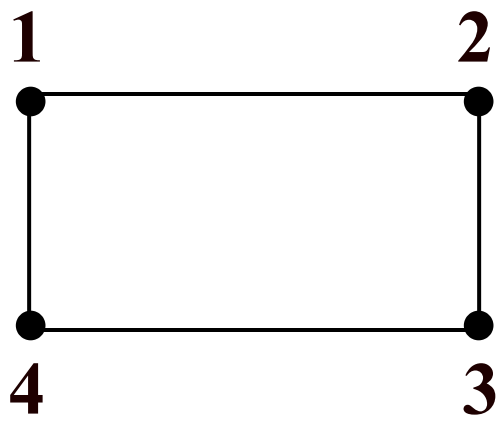
Reording the Rows/Columns

- 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:



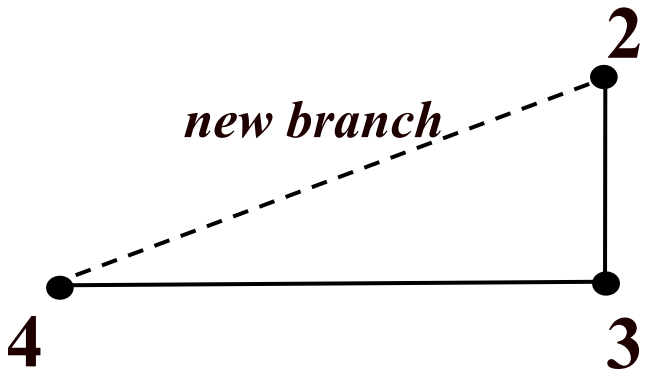
Reording Motivating Example

- 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



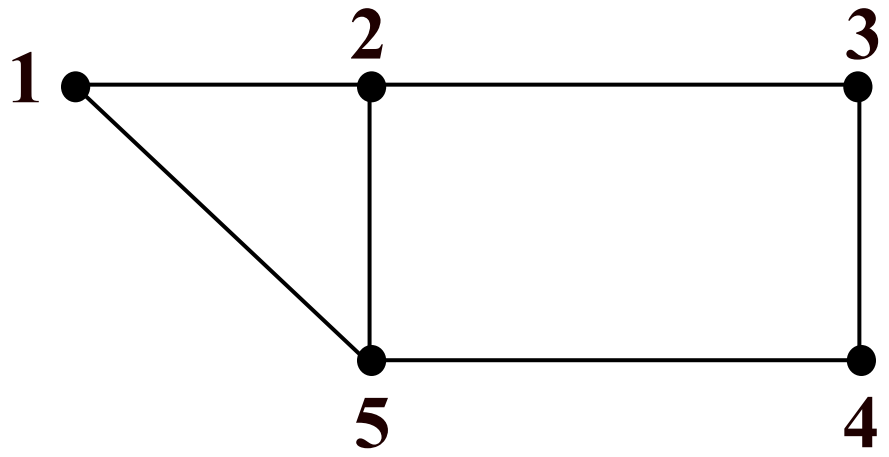
Reording Motivating Example

- 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

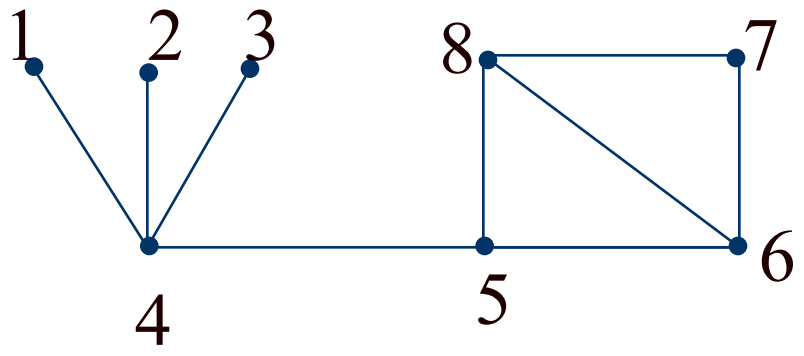
- There is no computationally efficient way to optimally reorder a sparse matrix; however, there are very efficient algorithms to greatly reduce the fills.
- Two steps here: 1) order the matrix, 2) add fills.
- A quite common algorithm combines ordering the matrix with adding the fills.
- The two methods 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

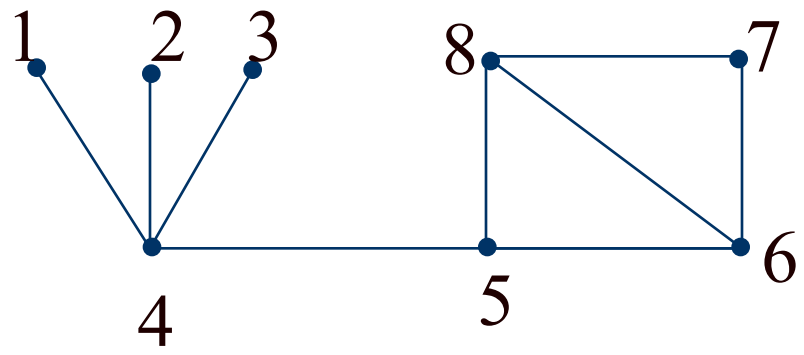
Number of new branches is 2 (4-8, 4-6)

Tinney Scheme 2

- The Tinney Scheme 2 usually combines adding the fills with the ordering in order to update the valence on-the-fly as the fills are added.
- As before the nodes are chosen based on their valence, but now the valence is the actual valence they have with the added lines (fills).
  - This is also known as the Minimum Degree Algorithm (MDA)
  - Ties are again broken using the lowest node number
- This method is quite effective for power systems, and is highly recommended; however, it is certainly not guaranteed to result in the fewest fills (i.e. not optimal).

Tinney Scheme 2 Example

- Consider the previous network:



- Nodes 1,2,3 are chosen as before. But after these nodes have been eliminated the valence of 4 is 1, so it is chosen next. Then 5 (with a new valence of 2 tied with 7), followed by 6 (new valence of 2), 7 then 8.

Coding Tinney 2

- The following slides show how to code Tinney 2 for an n by n sparse matrix **A**.
- First, we setup linked lists grouping all the nodes by their original valence
- vcHead is a pointer vector [0..mvValence].
  - If a node has no connections its incidence is 0
  - Theoretically mvValence should be n-1, but in practice a much smaller number can be used, putting nodes with valence values above this into the vcHead[mvValence] is

Coding Tinney 2, cont.

- Setup a boolean vectors chosenNode[1..n] to indicate which nodes are chosen and BSWR[1..n] as a sparse working row; initialize both to all false.
- Setup an integer vector rowPerm[1..n] to hold the permuted rows; initialize to all zeros.
- For i := 1 to n Do Begin
  - Choose node from valence data structure with the lowest current valence; let this be node k
    - Go through vcHead from lastchosen level (last chosen level may need to be reduced by one during the following elimination process;
  - Set rowPerm[i] = k; set chosenNode[k] = true

Coding Tinney 2, cont.

- Modify sparse matrix **A** to add fills between all of k's adjacent nodes provided
  1. a branch doesn't already exist
  2. both nodes have not already been chosen (their chosenNode entries are false)
  - These fills are added by going through each element in row k; for each element set the BSWR elements to true for the incident nodes; add fills if a connection does not already exist (this requires adding two new elements to **A**)
- Again go through row k updating the valence data structure for those nodes that have not yet been chosen
  - These values can either increase or go down by one (because of the elimination of node k)

- This continues through all the nodes; free all vectors except for rowPerm.
- At this point in the algorithm the rowPerm vector contains the new ordering and matrix **A** has been modified so that all the fills have been added.
  - The order of the rows in **A** has not been changed, and its columns are no longer sorted

Coding Tinney 2, cont.

- Sort the rows of **A** to match the order in rowPerm
  - Surprising, sorting **A** is of computational order equal to the number of elements in **A**
    - Go through **A** putting its elements into column linked lists; these columns will be ordered by row
    - Then through the columns linked lists in reverse order given by rowPerm
      - That is For i := n downto 1 Do Begin
        p1 := TSparmatLL(colHead[rowPerm[i]).Head;
        ….
- That's it – the matrix **A** is now readying for factoring.
- Pivoting may be required, but usually isn't needed in the power flow.

Some Example Values for Tinney 2

Number of buses	Nonzeros before fills	Fills	Total nonzeros	Percent nonzeros
37	63	72	135	9.86%
118	478	168	646	4.64%
18,190	64,948	31,478	96,426	0.029%
62,605	228,513	201,546	430,059	0.011%

Tinney Scheme 3

- "Number the rows so that at each step of the process the next row to be operated upon is the one that will introduce the fewest new nonzero terms."
- "If more than one row meets this criterion, select any one. This involves a trial simulation of every feasible alternative of the elimination process at each step. Input information is the same as for scheme 2)."
- Tinney 3 takes more computation and in general does not give fewer fills than the quicker Tinney 2

These are direct quotes from the Tinney-Walker 1967 IEEE Proceedings Paper

Sparse Forward Substitution with a Permutation Vector

Pass in **b** in bvector

```
For i := 1 to n Do Begin
  k = rowPerm[i];  // this is the only change, except using k
  p1 := rowHead[k];  // the row needs to be ordered correctly!
  While p1 <> rowDiag[k] Do Begin
    bvector[k] = bvector[k] – p1.value*bvector[p1.col];
    p1 := p1.next;
   End;
 End;
```

Sparse Backward Substitution with Permutation Vector

Pass in **b** in bvector

```
For i := n downto 1 Do Begin
  k = rowPerm[i];
  p1 := rowDiag[k].next;
  While p1 <> nil Do Begin
    bvector[k] = bvector[k] – p1.value*bvector[p1.col];
    p1 := p1.next;
   End;
   bvector[k] := bvector[k]/rowDiag[k].value;
 End;
```

- Note, numeric problems such as matrix singularity are indicated with rowDiag[k].value being zero!

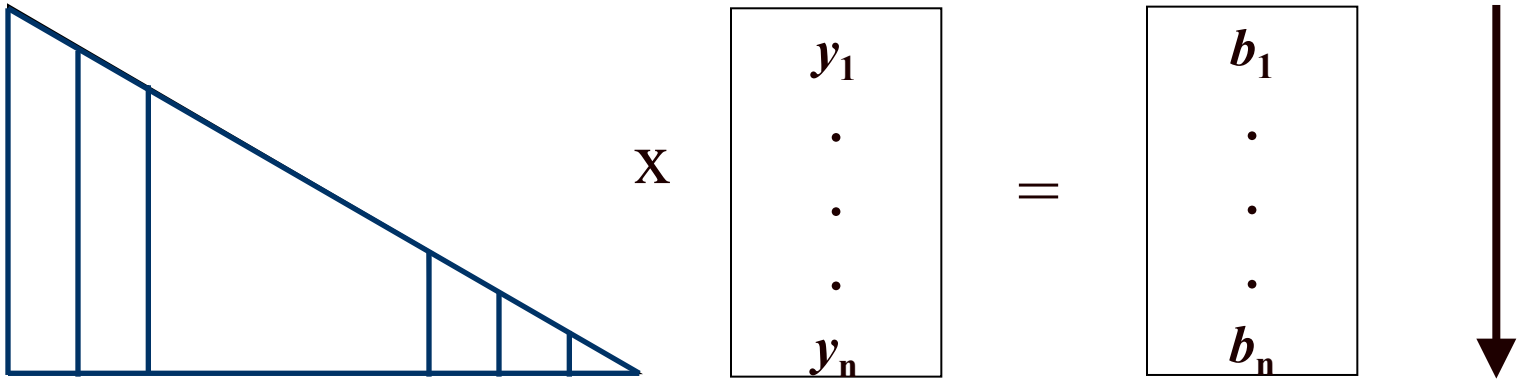
Sparse Vector Methods

- Sparse vector methods are useful for cases in solving **Ax**=**b** in which
  - **A** is sparse, **b** is sparse, only certain elements of **x** are needed
- In these right circumstances sparse vector methods can result in extremely fast solutions!
- A common example is to find selected elements of the inverse of **A**, such as diagonal elements.
- Often times multiple solutions with varying **b** values are required
  - **A** only needs to be factored once, with its factored form used many times
- Key reference is

W.F. Tinney, V. Brandwajn, and S.M. Chan, "Sparse Vector Methods", *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-104, no. 2, February 1985, pp. 295-300

Sparse Vector Methods Introduced

- Assume we are solving **Ax = b** with **A** factored so we solve **LUx = b** by first doing the forward substitution to solve **Ly = b** and then the backward substitution to solve **Ux = y**.
- A key insight: In the solution of **Ly = b** if **b** is sparse then only certain columns of **L** are required, and **y** is often sparse

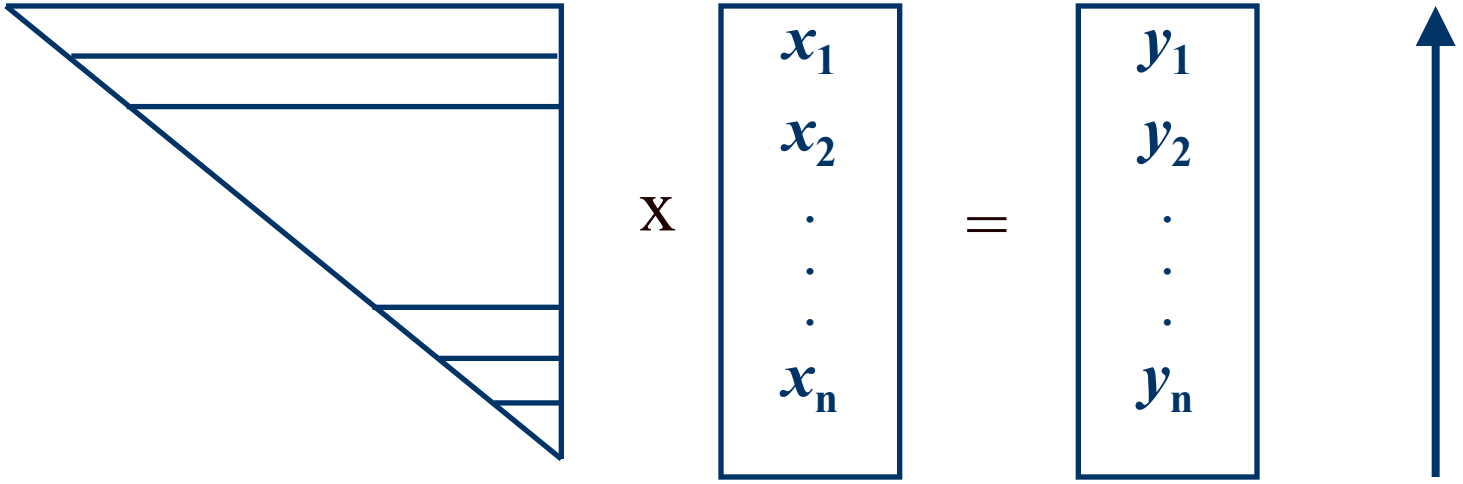


Fast Forward Substitution

- If **b** is sparse, then the fast forward (FF) substitution takes advantage of the fact that we only need certain columns of **L**
- We define {FF} as the set of columns of L needed for the solution of **Ly** = **b**; this is equal to the nonzero elements of **y**.
- In general, the solution of **Ux = y** will NOT result in **x** being a sparse vector.
- However, oftentimes only certain elements of **x** are desired
  - E.g., the sensitivity of the flows on certain lines to a change in generation at a single bus; or a diagonal of $\mathbf{A}^{-1}$

Fast Backward Substitution

- In the case in which only certain elements of **x** are desired, then we only need to use certain rows in **U** below the desired elements of **x**; define these columns as {FB}.
- This is known as a fast backward substitution (FB), which is used to replace the standard backward substitution.



Factorization Paths

- We observe that
  - {FF} depends on the sparsity structures of **L** and **b**
  - {FB} depends on the sparsity structures of **U** and **x**
- The idea of the factorization path provides a systematic way to construct these sets.
- A factorization path is an ordered set of nodes associated with the structure of the matrix.
- For FF the factorization path provides an ordered list of the columns of **L**.
- For FB the factorization path provides an ordered list of the rows of **U**.

Factorization Paths

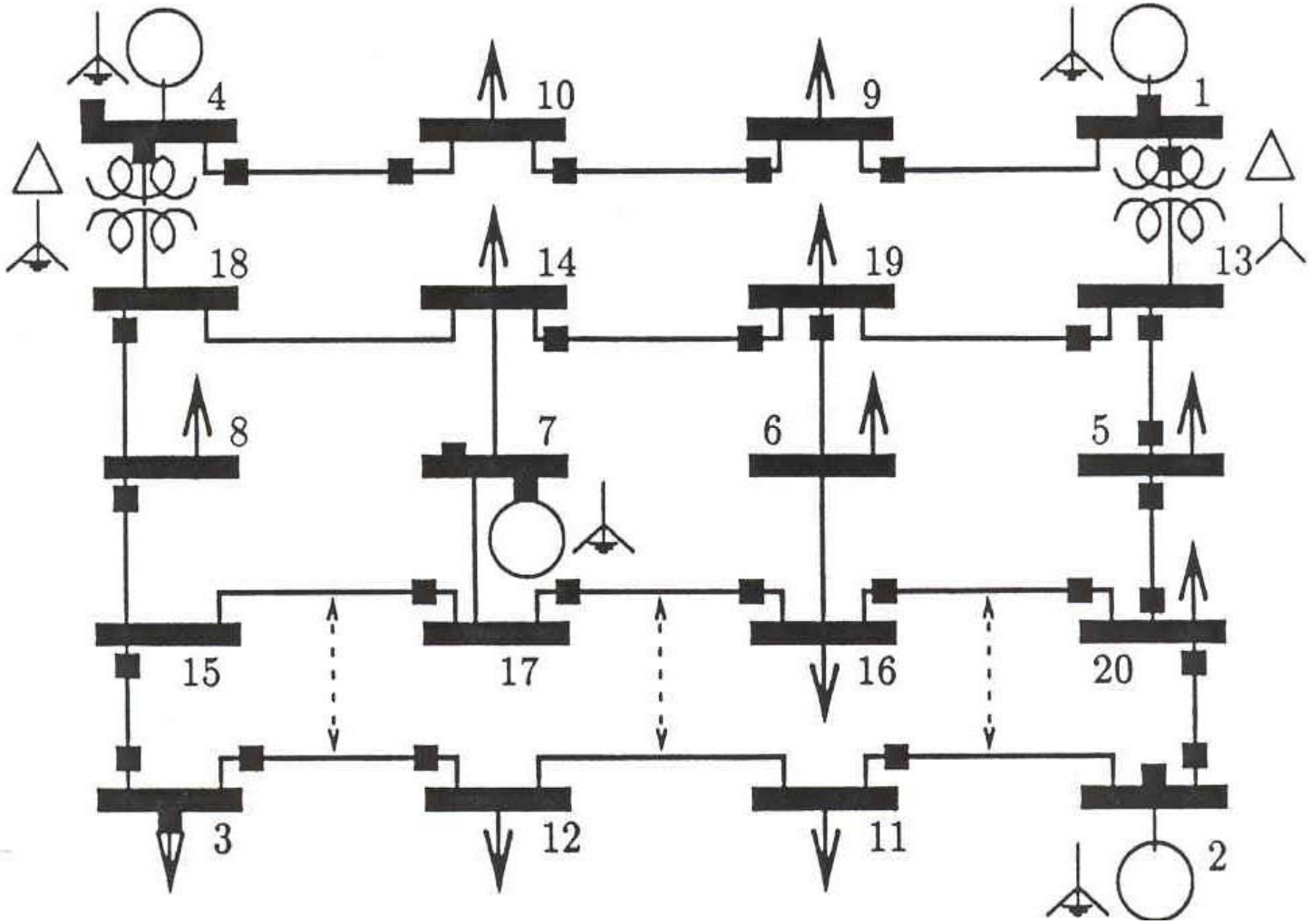
- The factorization path is traversed in the forward direction for FF and in the reverse direction for FB.
  - Factorization paths should be built using doubly linked lists
- A singleton vector is a vector with just one nonzero element. If this value is equal to one then it is a unit vector as well.
- With a sparse matrix structure ordered based upon the permutation vector order the path for a singleton with a now zero at position arow is build using the following code:

```
p1:= rowDiag[arow];
 While p1 <> nil Do Begin
  AddToPath(p1.col);   // Setup a doubly linked list!
  p1 := rowDiag[p1.col].next;
 End;
```

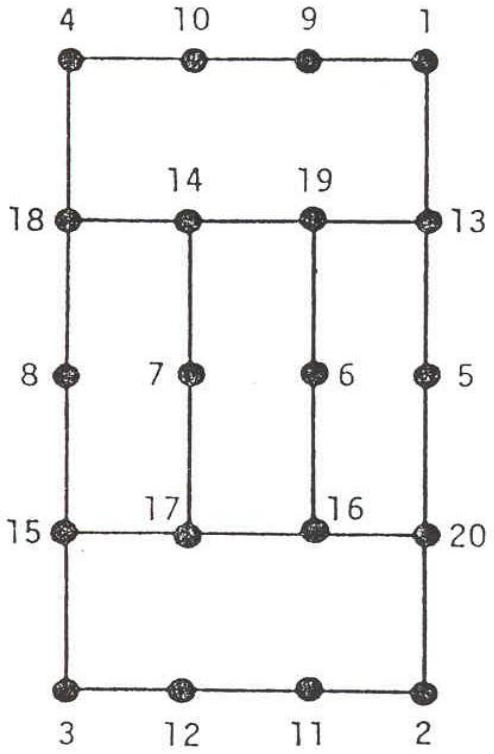
Path Table and Path Graph

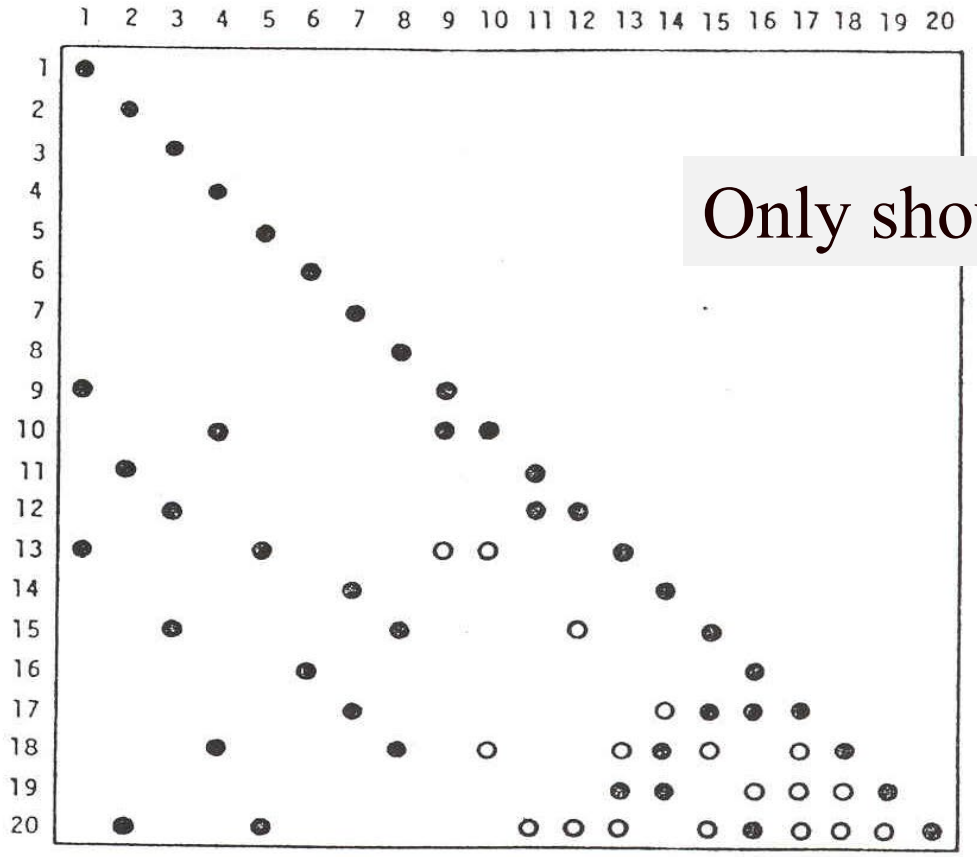
- The factorization path table is a vector that tells the next element in the factorization path for each row in the matrix.
- The factorization path graph shows a pictorial view of the path table.

20 Bus Example



20 Bus Example

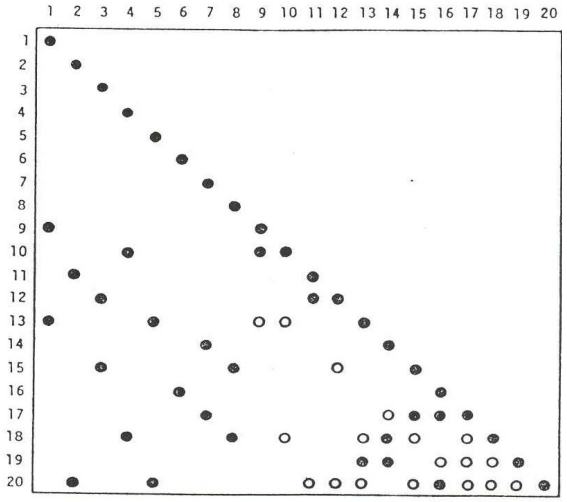


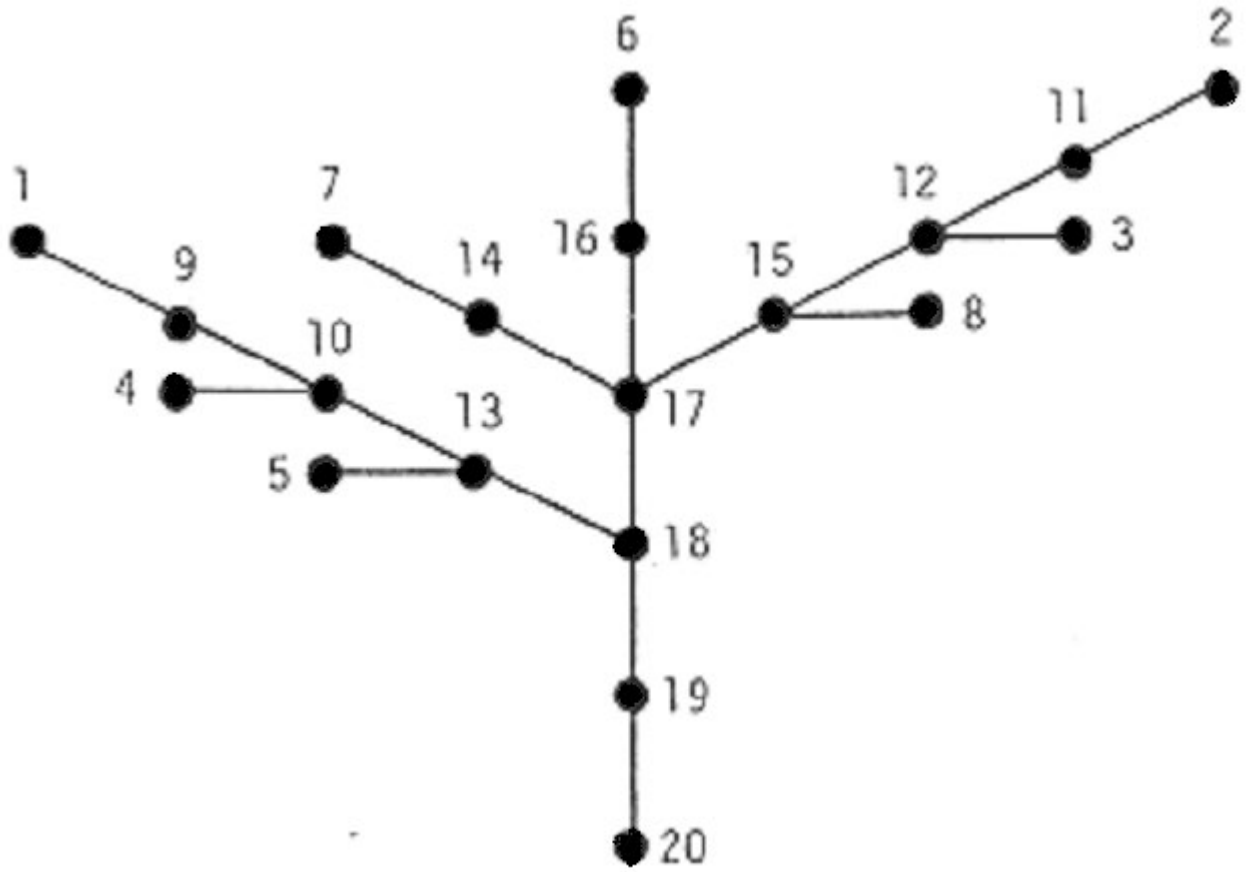


Only showing **L**

● Original Element (A and L)
○ Fill-In Element (L only)

20 Bus Example





NODE	NEXT	NODE	NEXT
1	9	11	12
2	11	12	15
3	12	13	18
4	10	14	17
5	13	15	17
6	16	16	17
7	14	17	18
8	15	18	19
9	10	19	20
10	13	20	0

20 Bus Example

- Suppose we wish to evaluate a sparse vector with the nonzero elements for components 2, 6, 7, and 12.
- From the path table or path graph, we obtain the following factorization paths (f.p.)

$$2 \rightarrow \textit{f.p.}\ \{2, 11, 12, 15, 17, 18, 19, 20\}$$

$$6 \rightarrow \textit{f.p.}\ \{6, 16, 17, 18, 19, 20\}$$

$$7 \rightarrow \textit{f.p.}\ \{7, 14, 17, 18, 19, 20\}$$

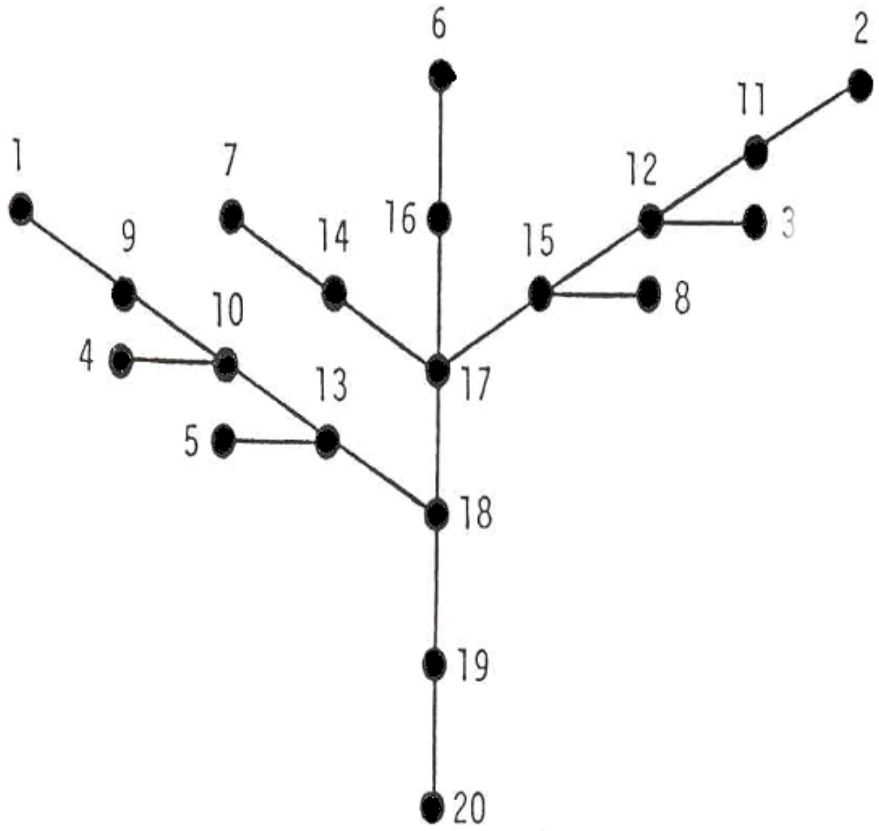
$$12 \rightarrow \textit{f.p. already contained in that for node 2}$$

- This gives the following path elements

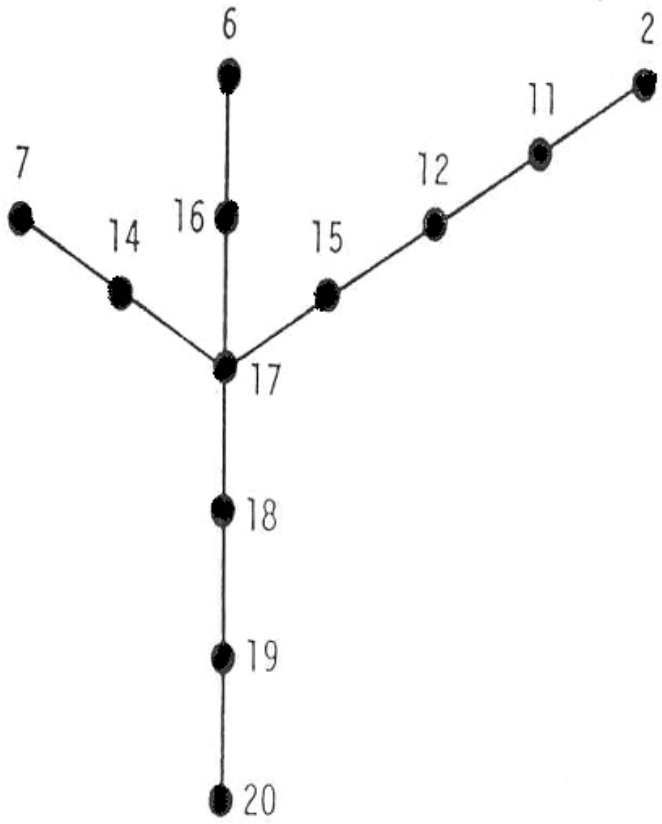
$$\{7, 14, 6, 16, 2, 11, 12, 15, 17, 18, 19, 20\}$$

20 Bus Example

Full path



Desired subset

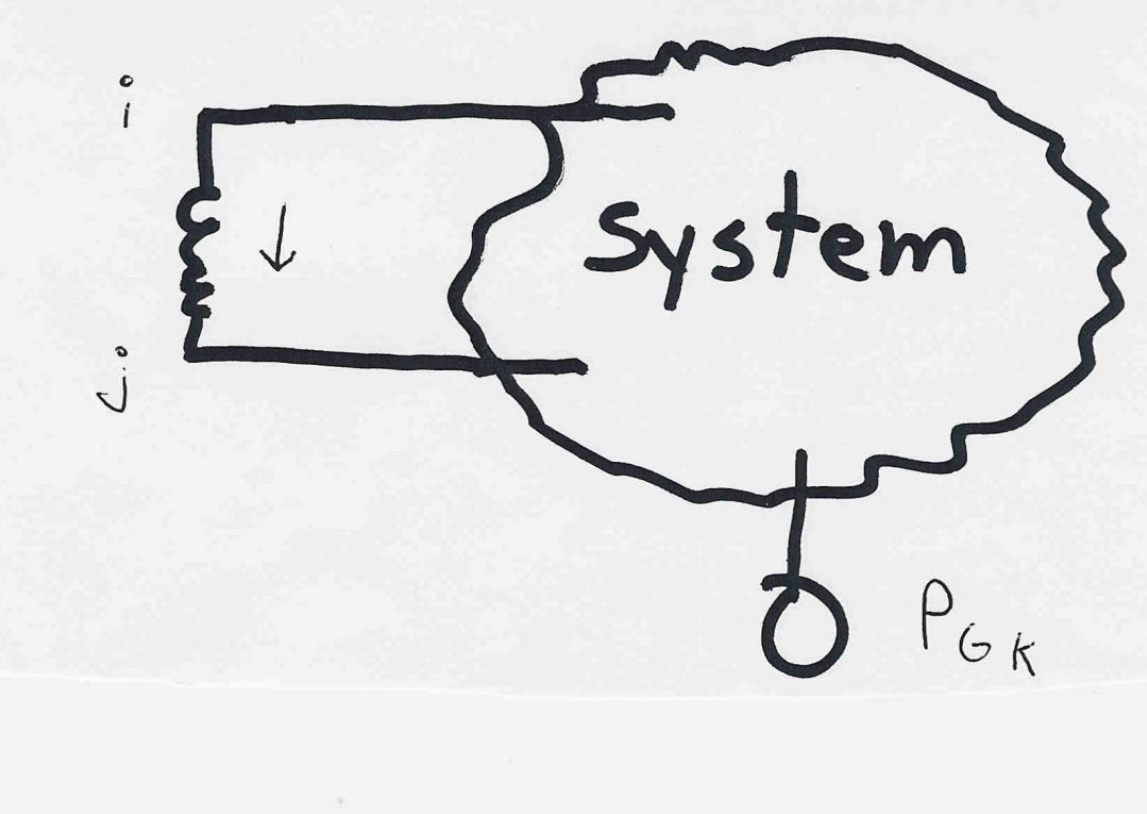


Power System Control and Sensitivities

- A major issue with power system operation is the limited capacity of the transmission system.
  - lines/transformers have limits (usually thermal)
  - no direct way of controlling flow down a transmission line (e.g., there are no valves to close to limit flow)
  - open transmission system access associated with industry restructuring is stressing the system in new ways
- We need to indirectly control transmission line flow by changing the generator outputs.
- Similar control issues with voltage.

Indirect Transmission Line Control

- What we would like to determine is how a change in generation at bus k affects the power flow on a line from bus i to bus j..



The assumption is that the change in generation is absorbed by the slack bus