

ECEN 615

Methods of Electric Power Systems Analysis

Lecture 10: Sparse Systems

Prof. Tom Overbye

Dept. of Electrical and Computer Engineering

Texas A&M University

overbye@tamu.edu



TEXAS A&M
UNIVERSITY

Announcements



- Read Chapter 7 from the book
- Homework 2 is due today
- Homework 3 should be done before the exam put does not need to be turned in
- First exam is Tuesday October 8 in class; closed book, closed notes. One 8.5 by 11 inch note sheet and calculators allowed

Example: 7 by 7 Matrix



- Consider the 7×7 matrix \mathbf{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



$r \backslash c$	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 original zero-nonzero structure

$r \backslash c$	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

The post-elimination zero nonzero pattern

Example: 7 by 7 Matrix Reordering



- We next reorder the rows and the columns of \mathbf{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 \mathbf{A}

Example: 7 by 7 Matrix Reordered Structure



$r \backslash c$	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

The reordered system

$r \backslash c$	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

The post-elimination reordered system

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

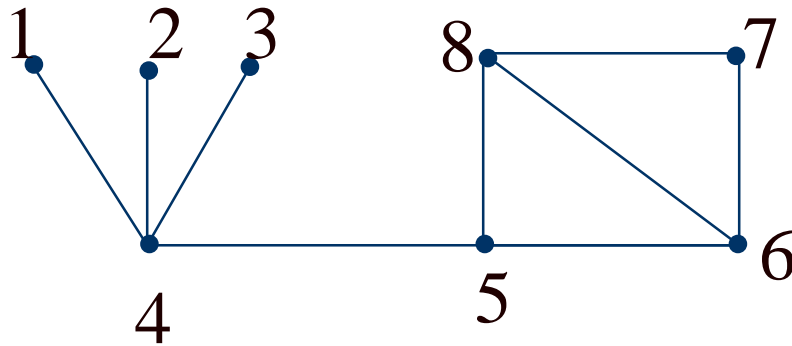


- Easy to describe, but not really used since the number of fills, while reduced, is still quite high
- In graph theory the degree (or valence or valency) of a vertex is the number of edges incident to the vertex
- Order the nodes (buses) by the number of incident branches (i.e., its valence) those with 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

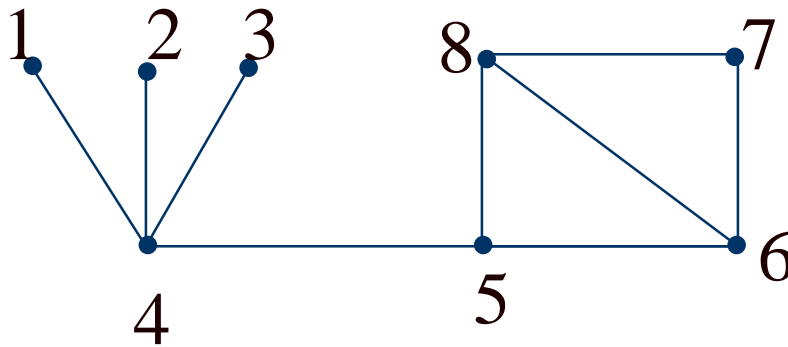


- The Tinney Scheme 2 usually combines adding the fills with the ordering in order to update the valence on-the-fly as the fills are added
- As before the nodes are chosen based on their valence, but now the valence is the actual valence they have with the added lines (fills)
 - This is also known as 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 once these nodes are 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 \mathbf{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 \mathbf{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 \mathbf{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)

Coding Tinney 2, cont.



- 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 \mathbf{A} has been modified so that all the fills have been added
 - The order of the rows in \mathbf{A} has not been changed, and its columns are no longer sorted

Coding Tinney 2, cont



- Sort the rows of \mathbf{A} to match the order in rowPerm
 - Surprising sorting \mathbf{A} is of computational order equal to the number of elements in \mathbf{A}
 - Go through \mathbf{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
 - $p1 := \text{TSparmatLL}(\text{colHead}[\text{rowPerm}[i]].\text{Head};$
 -
- That's it – the matrix \mathbf{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
- Tinney got into the NAE in 1998

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

$k = \text{rowPerm}[i];$

$p1 := \text{rowDiag}[k].\text{next};$

While $p1 \neq \text{nil}$ Do Begin

$\text{bvector}[k] = \text{bvector}[k] - p1.\text{value} * \text{bvector}[p1.\text{col}];$

$p1 := p1.\text{next};$

End;

$\text{bvector}[k] := \text{bvector}[k] / \text{rowDiag}[k].\text{value};$

End;

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

Sparse Vector Methods



- Sparse vector methods are useful for cases in solving $\mathbf{Ax}=\mathbf{b}$ in which
 - \mathbf{A} is sparse
 - \mathbf{b} is sparse
 - only certain elements of \mathbf{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 \mathbf{A} , such as diagonal elements.

Sparse Vector Methods



- Often times multiple solutions with varying \mathbf{b} values are required
 - \mathbf{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