ELEN 162
A. I. Zecevic

Lecture Notes for Week 9

Sparse Matrices

As we already mentioned, direct methods are primarily used in cases when matrix A is *sparse* (i.e. when the overwhelming majority of its elements are zero). Matrices of this sort are commonly encountered in engineering problems, because the number of physical connections between different system components is usually very limited. As a result, any given equation will typically involve only a few variables (even if the system size is very large).

We will now examine sparse matrices in some detail, and consider how their structure can be expoited to create efficient parallel algorithms.

Storage Techniques and Symbolic Factorization

One of the advantages of working with sparse matrices stems from the fact that we only need to store their nonzero elements. As a result, the necessary memory resources are usually quite modest. Although manipulating such matrices requires a certain amount of overhead (in the form of additional pointers and data structures), the added effort is clearly beneficial, since storing an entire $n \times n$ array can be very inefficient when n is large. We will not discuss the various techniques that were developed for this purpose at this point, but a brief overview is provided in the textbook.

Since we are primarily interested in solving system

$$Ax = b \tag{1}$$

using LU factorization, one of the biggest challenges that we must address is the fact that matrices L and U needn't necessarily be sparse (even if A itself is). Since the computational benefits of sparsity would be largely lost under such circumstances, it is necessary to develop methods that can minimize the number of nonzeros in L and U.

To see how this can be done, let us once again consider the "mechanics" of LU factorization, this time with the added assumption that matrix A is *stucturally symmetric* (which means that $a_{kj} \neq 0$ if and only if $a_{jk} \neq 0$). The approach that we will take in the following is somewhat different from the one we described previously, and is better suited for sparse matrices because it allows us to monitor the number and location of the additional nonzeros. Example 1 illustrates some of its main features.

Example 1. Consider the matrix

$$A = \begin{bmatrix} 5 & 1 & 2 \\ 1 & 4 & 1 \\ 2 & 2 & 5 \end{bmatrix} \tag{2}$$

and suppose that we wish to compute its factors

$$L = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \tag{3}$$

and

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{4}$$

As before, we will assume that $l_{11} = l_{22} = l_{33} = 1$, since the factorization is not unique. We now proceed to compute the elements of L and U in a step-by-step manner, starting with the first row of U and the first column of L.

STEP 1. To compute the first row of U, we will make use of the fact that

$$\begin{aligned} 5 = a_{11} = l_{11}u_{11} &\implies u_{11} = 5 \\ 1 = a_{12} = l_{11}u_{12} &\implies u_{12} = 1 \\ 2 = a_{13} = l_{11}u_{13} &\implies u_{13} = 2 \end{aligned} \tag{5}$$

We can now utilize the value that we obtained for u_{11} to determine the first column of L as

$$\begin{aligned} 1 = a_{21} = l_{21}u_{11} = 5l_{21} &\implies l_{21} = 0.2 \\ 2 = a_{31} = l_{31}u_{11} = 5l_{31} &\implies l_{31} = 0.4 \end{aligned} \tag{6}$$

After this step is completed, matrices L and U will have the form

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.2 & l_{22} & 0 \\ 0.4 & l_{32} & l_{33} \end{bmatrix} \tag{7}$$

and

$$U = \begin{bmatrix} 5 & 1 & 2 \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{8}$$

It is important to recognize at this point that the first row of U matches the first row of A exactly, and therefore has the *same* nonzero pattern. This holds true for the first column of L as well, since

$$l_{k1} = a_{k1}/u_{11} \tag{9}$$

As a result, $a_{k1} \neq 0$ obviously implies $l_{k1} \neq 0$.

This seemingly trivial observation will turn out to be very important, because it will allow us to anticipate the number of nonzero elements in matrices L and U without actually computing them. Such a procedure is known as *symbolic factorization*, and its benefits will be illustrated in subsequent examples.

STEP 2. In order to determine the second row of U and second column of L, we should observe that

$$\begin{aligned} a_{22} &= l_{21}u_{12} + l_{22}u_{22} \\ a_{23} &= l_{21}u_{13} + l_{22}u_{23} \\ a_{32} &= l_{31}u_{12} + l_{32}u_{22} \\ a_{33} &= l_{31}u_{13} + l_{32}u_{23} + l_{33}u_{33} \end{aligned} \tag{10}$$

Since we already know the terms on the right hand side that are associated with the first row of U (u_{12} and u_{13}) and first column of L (l_{21} and l_{31}), it makes sense to group them together with a_{22}, a_{23}, a_{32} and a_{33} (which are given). If we do so, (10) becomes

$$\begin{aligned} l_{22}u_{22} &= a_{22} - l_{21}u_{12} \\ l_{22}u_{23} &= a_{23} - l_{21}u_{13} \\ l_{32}u_{22} &= a_{32} - l_{31}u_{12} \\ l_{32}u_{23} + l_{33}u_{33} &= a_{33} - l_{31}u_{13} \end{aligned} \tag{11}$$

where all the unknown terms are on the left hand side.

Let us now define matrix

$$W_1 = \begin{bmatrix} l_{21}u_{12} & l_{21}u_{13} \\ l_{31}u_{12} & l_{31}u_{13} \end{bmatrix} \tag{12}$$

which is formed from the off-diagonal elements in column 1 of L and row 1 of U. It is not difficult to see that this matrix can be equivalently represented as

$$W_1 = \begin{bmatrix} l_{21} \\ l_{31} \end{bmatrix} \cdot \begin{bmatrix} u_{12} & u_{13} \end{bmatrix} \tag{13}$$

which will be more convenient for our purposes. The reason for introducing matrix W_1 is that it allows us to express the right hand side of (11) in matrix form as

$$\begin{bmatrix} (a_{22} - l_{21}u_{12}) & (a_{23} - l_{21}u_{13}) \\ (a_{32} - l_{31}u_{12}) & (a_{33} - l_{31}u_{13}) \end{bmatrix} = F_1 - W_1 \tag{14}$$

where

$$F_1 = \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} \tag{15}$$

Setting

$$A_2 = \begin{bmatrix} l_{22}u_{22} & l_{22}u_{33} \\ l_{32}u_{22} & l_{32}u_{23} + l_{33}u_{33} \end{bmatrix} = \begin{bmatrix} l_{22} & 0 \\ l_{32} & l_{33} \end{bmatrix} \cdot \begin{bmatrix} u_{22} & u_{23} \\ 0 & u_{33} \end{bmatrix} \tag{16}$$

we can now rewrite (11) as

$$A_2 = F_1 - W_1 \tag{17}$$

Expression (17) tells us that the remaining elements of L and U can be computed by factorizing matrix A_2, which is known at this point. For the sake of clarity, in the following we will denote the elements of this matrix as

$$A_2 = \begin{bmatrix} a_{22}^{(2)} & a_{23}^{(2)} \\ a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix} \tag{18}$$

to emphasize the fact that they differ from the original elements a_{22}, a_{23}, a_{32}, and a_{33}.

The simple transformation described above reduces the problem to factorizing a matrix that is smaller than the original one. Given that

$$F_1 = \begin{bmatrix} 4 & 1 \\ 2 & 5 \end{bmatrix} \tag{19}$$

this matrix has the form

$$A_2 = F_1 - W_1 = \begin{bmatrix} 4 & 1 \\ 2 & 5 \end{bmatrix} - \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 3.8 & 0.6 \\ 1.6 & 4.2 \end{bmatrix} \tag{20}$$

We can now perform Step 1 on A_2 to compute the second row of U and the second column of L. If we follow the same procedure as before, we obtain

$$\begin{aligned} u_{22} &= a_{22}^{(2)} = 3.8 \\ u_{23} &= a_{23}^{(2)} = 0.6 \end{aligned} \tag{21}$$

and

$$\begin{aligned} l_{22} &= 1 \\ l_{32} &= a_{32}^{(2)}/u_{22} = 1.6/3.8 = 0.421 \end{aligned} \tag{22}$$

STEP 3. After Step 2 has been completed, L and U become

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & 0.421 & l_{33} \end{bmatrix} \tag{23}$$

and

$$U = \begin{bmatrix} 5 & 1 & 2 \\ 0 & 3.8 & 0.6 \\ 0 & 0 & u_{33} \end{bmatrix} \tag{24}$$

respectively. The only element that remains to be computed at his point is u_{33} (since $l_{33} = 1$ by assumption). In order to do that, we should recall that $a_{33}^{(2)} = a_{33} - l_{31}u_{13}$ (this follows directly from equation (14), (17) ad (18)). With that in mind, we can rewrite the last equation in (10) as

$$l_{33}u_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23} = a_{33}^{(2)} - l_{32}u_{23} \tag{25}$$

Setting $F_2 = a_{33}^{(2)}$ and $W_2 = l_{32}u_{23}$, we can determine u_{33} by factorizing matrix

$$A_3 = F_2 - W_2 = a_{33}^{(2)} - l_{32}u_{23} = 4.2 - 0.421 \cdot 0.6 = 3.947 \tag{26}$$

This is obviously a 1×1 matrix whose factorization has the form

$$A_3 = l_{33}u_{33} \tag{27}$$

Recalling that $l_{33} = 1$, we now easily obtain $u_{33} = 3.947$, and matrices L and U assume their final form

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.4 & 0.421 & 1 \end{bmatrix} \tag{28}$$

and

$$U = \begin{bmatrix} 5 & 1 & 2 \\ 0 & 3.8 & 0.6 \\ 0 & 0 & 3.947 \end{bmatrix} \tag{29}$$

One of the most important features of the algorithm that we just described is that fact that it recursively applies Step 1 to matrices A_2, A_3, etc. In each stage of this process, the matrix size is reduced by 1, and A_{k+1} can be computed as

$$A_{k+1} = F_k - l_k \cdot u_k \tag{30}$$

where

$$l_k = \begin{bmatrix} l_{k+1,k} \\ l_{k+2,k} \\ \vdots \\ l_{n,k} \end{bmatrix} \tag{31}$$

and

$$u_k = \begin{bmatrix} u_{k,k+1} & u_{k,k+2} & \cdots & u_{k,n} \end{bmatrix} \tag{32}$$

represent the *off-diagonal elements* in column k of matrix L and row k of matrix U.

To see why this is useful, suppose that we have performed k steps of the factorization, and are now in the process of computing matrix A_{k+1}. In general, this matrix will have more nonzeros than matrix A_k, since the product $l_k \cdot u_k$ can add new elements to F_k. What is less obvious (but equally important) is that we can predict their number and location directly from the structure of l_k and u_k without actually computing their values. We will see that this procedure (which is known as symbolic factorization) is an essential part of several algorithms for sparsity preservation.

The diagram in Fig. 1 indicates how symbolic factorization works if l_k has nonzeros in rows i, j and m (note that u_k will have the same nonzero pattern, since A is assumed to be structurally symmetric).

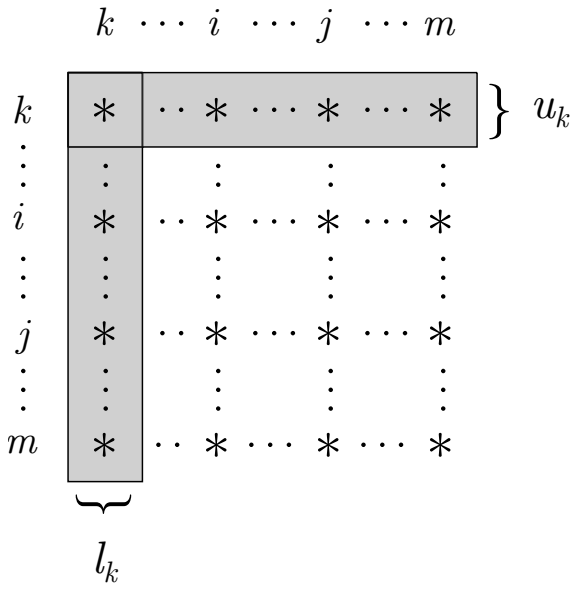

Figure 1: A schematic representation of symbolic factorization.

It is not difficult to see that the product $l_k \cdot u_k$ introduces new elements in locations (i,i), (i,j), (i,m), (j,i), (j,j), (j,m), (m,i), (m,j), and (m,m). Some of these elements will be

nonzeros in A_{k+1}, even though their value was zero in matrix A_k. For this reason, they are commonly referred to as "fill-ins", and minimizing their number is critical for preserving sparsity.

The following example illustrates why fill-in reduction is so important in solving systems of linear equations.

Example 2. Consider the matrix

$$A = \begin{bmatrix} * & * & * & * & * \\ * & * & 0 & 0 & 0 \\ * & 0 & * & 0 & 0 \\ * & 0 & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix} \tag{33}$$

whose nonzero elements are indicated by $*$. After the first step of LU factorization, we obtain

$$A_2 = F_1 - l_1 \cdot u_1 \tag{34}$$

where

$$F_1 = \begin{bmatrix} * & 0 & 0 & 0 \\ 0 & * & 0 & 0 \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & * \end{bmatrix} \tag{35}$$

and l_1 and u_1 have the structure

$$l_1 = \begin{bmatrix} * \\ * \\ * \\ * \end{bmatrix} \tag{36}$$

and

$$u_1 = \begin{bmatrix} * & * & * & * \end{bmatrix} \tag{37}$$

(since they must match the nonzero pattern of row 1 and column 1 of A).

Given that the product $l_1 \cdot u_1$ gives rise to a *full* 4×4 matrix, the nonzero pattern of A_2 will be

$$A_2 = \begin{bmatrix} * & \odot & \odot & \odot \\ \odot & * & \odot & \odot \\ \odot & \odot & * & \odot \\ \odot & \odot & \odot & * \end{bmatrix} \tag{38}$$

where the symbol $\odot$ denotes fill-ins. This implies that L and U will have the form

$$L = \begin{bmatrix} * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 \\ * & \odot & * & 0 & 0 \\ * & \odot & \odot & * & 0 \\ * & \odot & \odot & \odot & * \end{bmatrix} \tag{39}$$

and

$$U = \begin{bmatrix} * & * & * & * & * \\ 0 & * & \odot & \odot & \odot \\ 0 & 0 & * & \odot & \odot \\ 0 & 0 & 0 & * & \odot \\ 0 & 0 & 0 & 0 & * \end{bmatrix} \tag{40}$$

It is not difficult to see that all elements that were zero in matrix A are now nonzeros in L and U. We can therefore conclude that the original sparsity of A is *completely lost* in the factorization process.

What can be done to mitigate this problem, and reduce the accumulation of fill-in elements? The following example illustrates a simple idea that we will develop further once we introduce the concept of an elimination graph.

Example 3. Suppose that we permute the matrix given in (33) by swapping rows 1 and 5 (and doing the same with columns 1 and 5). Such a permutation is said to be *symmetric*, and can be described by a single permutation vector $p = [5\ 2\ 3\ 4\ 1]$. The resulting matrix will then have the form

$$\tilde{A} = \begin{bmatrix} * & 0 & 0 & 0 & * \\ 0 & * & 0 & 0 & * \\ 0 & 0 & * & 0 & * \\ 0 & 0 & 0 & * & * \\ * & * & * & * & * \end{bmatrix} \tag{41}$$

and the first step of LU factorization will produce

$$\tilde{A}_2 = \tilde{F}_1 - l_1 \cdot u_1 \tag{42}$$

where

$$\tilde{F}_1 = \begin{bmatrix} * & 0 & 0 & * \\ 0 & * & 0 & * \\ 0 & 0 & * & * \\ * & * & * & * \end{bmatrix} \tag{43}$$

and l_1 and u_1 have the form

$$l_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ * \end{bmatrix} \tag{44}$$

and

$$u_1 = \begin{bmatrix} 0 & 0 & 0 & * \end{bmatrix} \tag{45}$$

respectively. Since

$$\tilde{A}_2 = \tilde{F}_1 - l_1 \cdot u_1 = \begin{bmatrix} * & 0 & 0 & * \\ 0 & * & 0 & * \\ 0 & 0 & * & * \\ * & * & * & * \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & * \end{bmatrix} \tag{46}$$

it follows that matrices $\tilde{A}_2$ and $\tilde{F}_1$ will have the *same* nonzero pattern (although one of their elements will have different values).

If we now execute the remaining steps in this process, it is easily verified that the factorization produces *no fill-ins at all*, and that matrices L and U will have the form

$$L = \begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & * & 0 \\ * & * & * & * & * \end{bmatrix} \tag{47}$$

and

$$U = \begin{bmatrix} * & 0 & 0 & 0 & * \\ 0 & * & 0 & 0 & * \\ 0 & 0 & * & 0 & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix} \tag{48}$$

This suggests that an appropriately chosen permutation can help preserve the sparsity of the original matrix, and can dramatically reduce the number of operations needed to compute matrices L and U. We will now explain how such a permutation can be found.

Elimination Graphs

Algorithms for symbolic factorization rely on the fact that any structurally symmetric matrix A can be uniquely associated with an *undirected graph* in which vertices i and j are connected if and only if $a_{ij} \neq 0$ (which automatically implies $a_{ji} \neq 0$ as well). In such a graph, edges will obviously correspond to nonzero elements.

The following example illustrates how this idea can be applied to monitor fill-in accumulation in the process of LU factorization. Since the graph changes in each step, it is commonly referred to as an *elimination graph*.

Example 4. Consider a matrix A whose nonzero structure is

$$A = \begin{bmatrix} * & * & * & * & 0 \\ * & * & 0 & 0 & * \\ * & 0 & * & 0 & 0 \\ * & 0 & 0 & * & 0 \\ 0 & * & 0 & 0 & * \end{bmatrix} \tag{49}$$

This matrix can be represented as an undirected graph in the manner shown in Fig. 2. Since the first column of matrix L and the first row of matrix U have exactly the same nonzero pattern as matrix A, vectors l_1 and u_1 will have the form

$$l_1 = \begin{bmatrix} * \\ * \\ * \\ 0 \end{bmatrix} \tag{50}$$

and

$$u_1 = \begin{bmatrix} * & * & * & 0 \end{bmatrix} \tag{51}$$

respectively.

Observing that

$$F_1 = \begin{bmatrix} * & 0 & 0 & * \\ 0 & * & 0 & 0 \\ 0 & 0 & * & 0 \\ * & 0 & 0 & * \end{bmatrix} \tag{52}$$

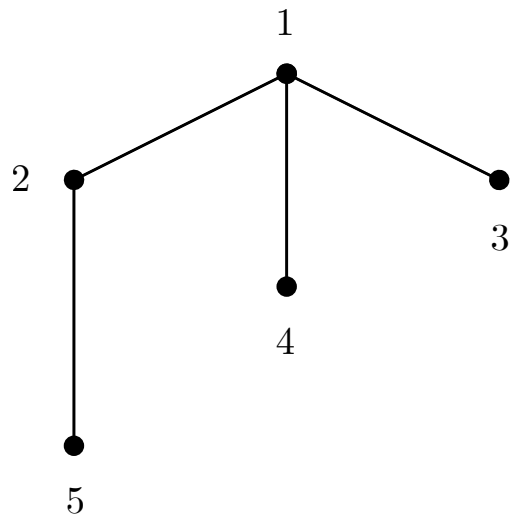

Figure 2: Graph that corresponds to the matrix in (49).

it follows that

$$A_2 = F_1 - l_1 \cdot u_1 = \begin{bmatrix} * & \odot & \odot & * \\ \odot & * & \odot & 0 \\ \odot & \odot & * & 0 \\ * & 0 & 0 & * \end{bmatrix} \tag{53}$$

The graph that corresponds to matrix A_2 is shown in Fig. 3, in which the dashed lines represent the fill-ins that have been added in this step.

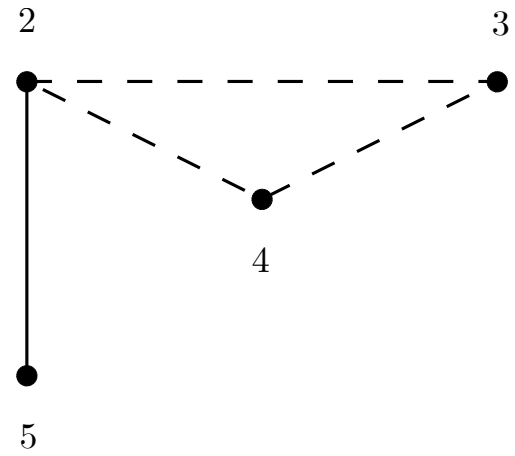

Figure 3: The graph that corresponds to matrix A_2.

It is important to recognize, however, that we could have obtained this graph *without actually forming matrix* A_2. We could have done so by utilizing the fact that all three neighbors of node 1 (which are nodes 2, 3 and 4) become *pairwise connected* after the product $l_1 \cdot u_1$ is subtracted from matrix F_1. This would allow us to form the graph that corresponds to A_2 by eliminating node 1, removing all edges that are incident to it, and connecting all of its neighbors. Any new edges that arise in this process constitute fill-ins, and are indicated by dashed lines (in this case, the fill-ins are $(2,3)$, $(2,4)$ and $(3,4)$).

If we adopt this procedure for monitoring fill-in, our next step should start with the graph in Fig. 3 (which corresponds to A_2).We can then form matrix A_3 by eliminating node 2 and removing all the edges that are incident to it. After connecting all of its neighbors (which happen to be nodes 3, 4 and 5), we obtain the graph shown in Fig. 4, in which the dashed lines once again indicate fill-ins (these elements occur in positions $(3, 5)$ and $(4, 5)$).

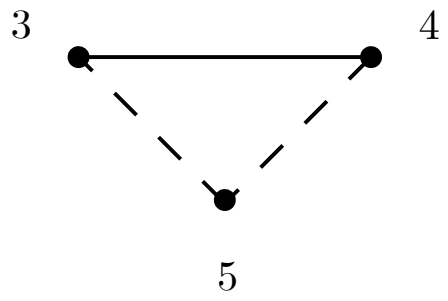

Figure 4: The graph after eliminating node 2.

In the graph that correspond to A_3, nodes 4 and 5 (which are the neighbors of node 3) are already connected. As a result, removing node 3 produces the graph shown in Fig. 5, in which there are no dashed lines. This indicates that no fill-ins were created in the current step.

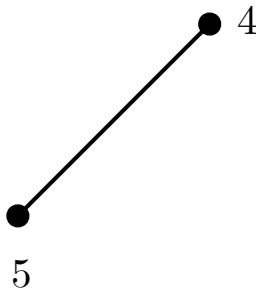

Figure 5: The graph after eliminating node 3.

In the final step, we need to remove node 4 and all of its incident edges, which leaves us with node 5. Given that this was the only neighbor of node 4, there is nothing to connect at this point, and there are no new fill-ins. We can therefore conclude that L and U will have the form

$$L = \begin{bmatrix} * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 \\ * & \odot & * & 0 & 0 \\ * & \odot & \odot & * & 0 \\ 0 & * & \odot & \odot & * \end{bmatrix} \tag{54}$$

and

$$U = \begin{bmatrix} * & * & * & * & 0 \\ 0 & * & \odot & \odot & * \\ 0 & 0 & * & \odot & \odot \\ 0 & 0 & 0 & * & \odot \\ 0 & 0 & 0 & 0 & * \end{bmatrix} \tag{55}$$

The following example (which features a somewhat larger matrix) further underscores the practical value of using elimination graphs for monitoring the number and location of fill-in elements in matrices L and U.

Example 5. Consider the matrix

$$A = \begin{bmatrix} * & * & * & * & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & 0 & * & 0 & * & * & 0 & 0 \\ * & 0 & 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 0 & * & 0 & * & 0 \\ 0 & 0 & * & 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \tag{56}$$

whose graph-theoretic representation is provided in Fig. 6. If we proceed to eliminate the nodes in ascending order, the elimination graph will evolve in the manner shown in Figs. 7-12. From these figures, we can easily conclude that matrices L and U have the form

$$L = \begin{bmatrix} * & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & \odot & * & 0 & 0 & 0 & 0 & 0 \\ * & \odot & \odot & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & \odot & * & 0 & 0 & 0 \\ 0 & 0 & * & \odot & \odot & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & \odot & * & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \tag{57}$$

and

$$U = \begin{bmatrix} * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & \odot & \odot & 0 & 0 & 0 & 0 \\ 0 & 0 & * & \odot & * & * & 0 & 0 \\ 0 & 0 & 0 & * & \odot & \odot & 0 & 0 \\ 0 & 0 & 0 & 0 & * & \odot & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & \odot & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \tag{58}$$

with 7 fill-in elements in each case.

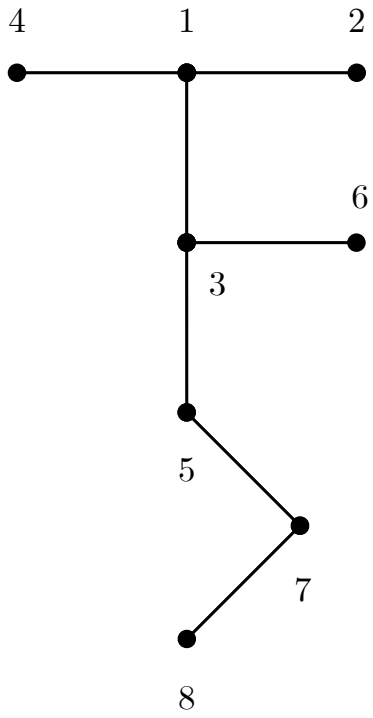

Figure 6: The graph that corresponds to the matrix in (56).

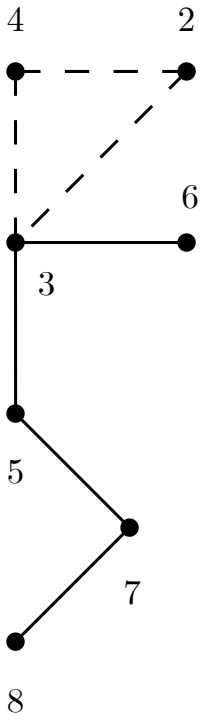

Figure 7: Graph after removing node 1 (its neighbors are $\{2, 3, 4\}$). New fill-ins are $(2, 3)$, $(2, 4)$ and $(3, 4)$.

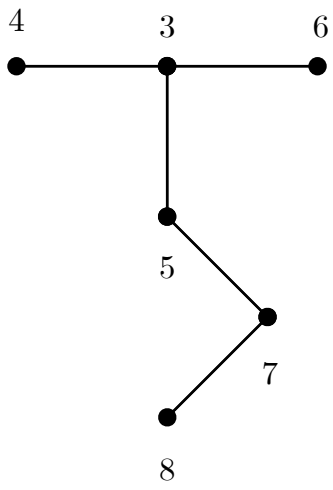

Figure 8: Graph after removing node 2 (its neighbors are $\{3, 4\}$). No new fill-ins.

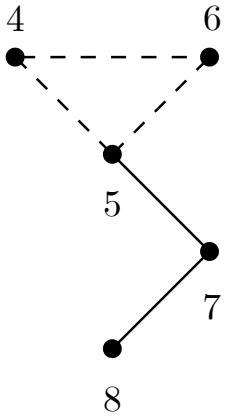

Figure 9: Graph after removing node 3 (its neighbors are $\{4, 5, 6\}$). New fill-ins are $(4, 5)$, $(4, 6)$ and $(5, 6)$.

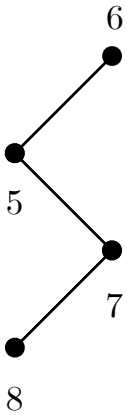

Figure 10: Graph after removing node 4 (its neighbors are $\{5, 6\}$). No new fill-ins.

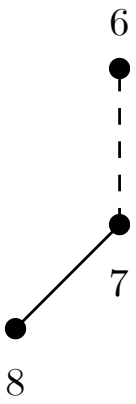

Figure 11: Graph after removing node 5 (its neighbors are $\{6, 7\}$). New fill-ins: $(6, 7)$.

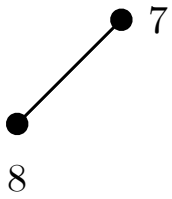

Figure 12: Graph after removing node 6 (its neighbor is $\{7\}$). No new fill-ins.

The Minimal Degree Ordering

The minimal degree ordering is one of the most successful techniques for reducing fill-in that arises in the process of LU factorization. This approach utilizes elimination graphs and the procedure outlined in Examples 4 and 5 to systematically identify a sparsity-preserving permutation of matrix A.

To see how this can be done, we should first observe that fill-ins are more likely to appear if the node that we are eliminating has a large number of neighbors. It therefore makes sense to eliminate the node that has the *smallest* degree in any given step (instead of eliminating them in ascending order). In the case of a tie, we will always choose the lowest numbered node (for the sake of simplicity). This approach isn't necessarily optimal but is easy to implement, which gives it certain practical advantages.

The following example (which involves the same matrix that we considered in Example 5) illustrates the effectiveness of such a strategy.

Example 6. Since we will be working with matrix (56) one again, our starting point will be the graph shown in Fig. 6. Although nodes 2, 4, 6 and 8 have the same degree in this graph, the tie breaking criterion that we adopted identifies node 2 as the one that should be removed first. The resulting graph is shown in Fig. 13, and subsequent stages of this process are described in Figs. 14-18.

The order in which the nodes were eliminated uniquely defines a permutation that reduces fill-in. In this particular case the corresponding permutation vector

$$p = \begin{bmatrix} 2 & 4 & 1 & 6 & 3 & 5 & 7 & 8 \end{bmatrix} \tag{59}$$

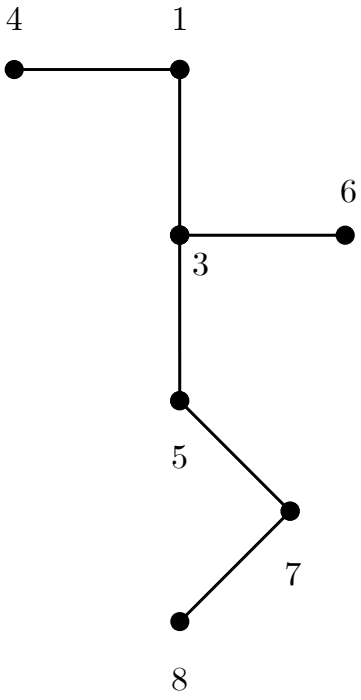

Figure 13: Graph after removing node 2 (its neighbor is {1}). No new fill-ins.

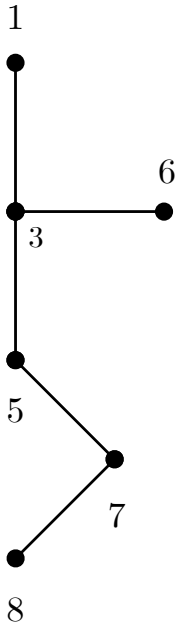

Figure 14: Graph after removing node 4 (its neighbor is {1}). No new fill-ins.

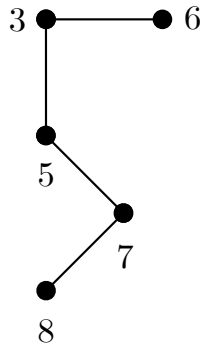

Figure 15: Graph after removing node 6 (its neighbor is {3}). No new fill-ins.

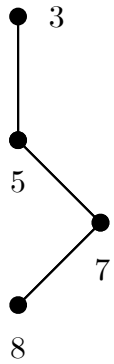

Figure 16: Graph after removing node 1 (its neighbor is {3}). No new fill-ins.

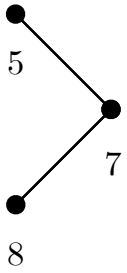

Figure 17: Graph after removing node 3 (its neighbor is {5}). No new fill-ins.

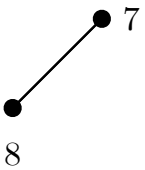

Figure 18: Graph after removing node 5 (its neighbor is {7}). No new fill-ins.

tells us precisely how the rows and columns of matrix A should be rearranged. When we do so, we obtain a matrix $\tilde{A}$ whose structure is

$$\tilde{A} = \begin{bmatrix} * & 0 & 0 & * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 0 & * & 0 & 0 & 0 \\ * & * & 0 & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \tag{60}$$

Figures 13-18 indicate that factorizing this matrix produces *no fill-in at all*, which means that L and U will have the *same* nonzero pattern as the lower and upper triangular parts of $\tilde{A}$. This is a significant improvement over the result that we obtained in Example 5, where matrix A was *not* permuted (in that case, we had a total of 14 fill-in elements).

It goes without saying, of course, that fill-in can rarely be completely eliminated in practical problems, even if the original matrix is very sparse. Nevertheless, this example nicely illustrates how a simple reordering scheme can drastically reduce the computational effort needed to solve systems of linear equations.

Decomposition Algorithms

Although the minimal degree ordering is very effective when it comes to minimizing the amount of fill-in, the permuted matrix that it produces usually lacks structure. As a result, there is no obvious way to parallelize the computation of matrices L and U. With that in mind, in this section we will consider an alternative approach which reorders the matrix into a *bordered block diagonal* (BBD) form (like the one shown in Fig. 19).

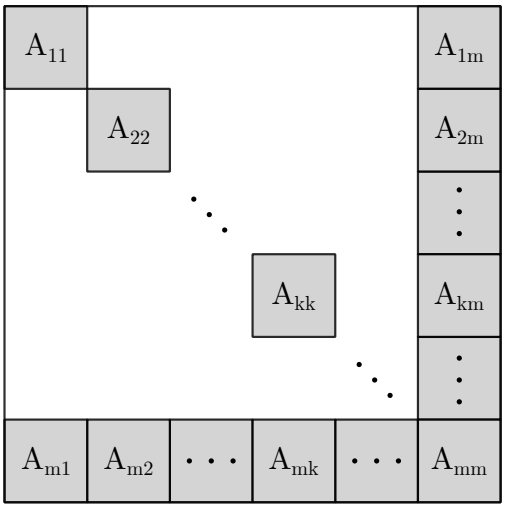

Figure 19: A matrix with a BBD structure.

To get a sense for why the BBD form is suitable for parallelization, we should first observe that matrices L and U which correspond to it have the form shown in Figs. 20 and 21.

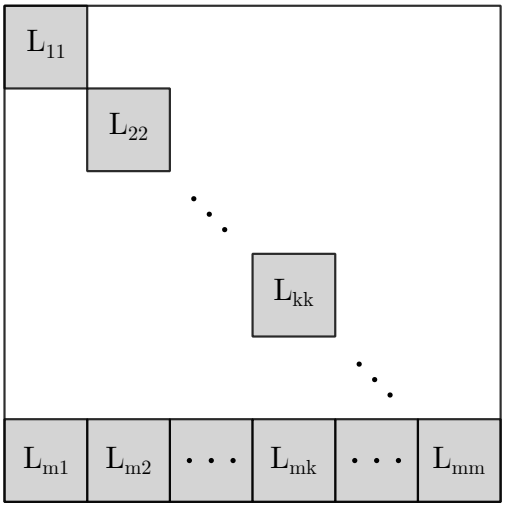

Figure 20: The structure of matrix L.

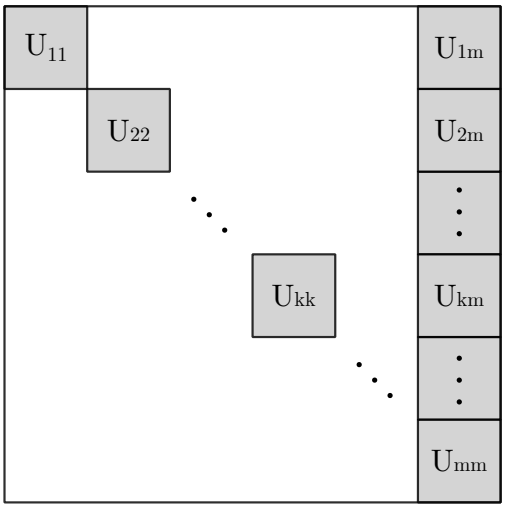

Figure 21: The structure of matrix U.

If we multiply these two matrices, it is easily verified that the blocks of A can be expressed as

$$A_{kk} = L_{kk}U_{kk} \tag{61}$$

$$A_{km} = L_{kk}U_{km} \tag{62}$$

and

$$A_{mk} = L_{mk}U_{kk} \tag{63}$$

for $k = 1, 2, \ldots, m-1$. The only block that *does not* conform to this simple pattern is A_{mm}, since it corresponds to the sum

$$A_{mm} = \sum_{k=1}^{m} L_{mk}U_{km} \tag{64}$$

Expression (61) indicates that L_{kk} and U_{kk} can be computed by factorizing matrix A_{kk}. Once L_{kk} and U_{kk} are known, U_{km} can be determined by solving the system

$$L_{kk}U_{km} = A_{km} \tag{65}$$

This is not difficult to do, since (65) represents a collection of equations of the form

$$L_{kk}u_i = a_i \tag{66}$$

where u_i and a_i represent the i-th columns of matrices U_{km} and A_{km}, respectively.

A similar procedure can be used to compute matrix L_{mk} as well, since

$$A_{mk} = L_{mk}U_{kk} \tag{67}$$

The only modification that needs to be made in this case involves rewriting expression (67) as

$$U_{kk}^T L_{mk}^T = A_{mk}^T \tag{68}$$

This allows us to determine matrix L_{mk} by solving multiple systems of the form

$$U_{kk}^T x_i = w_i \tag{69}$$

where x_i and w_i denote columns of matrices L_{mk}^T and A_{mk}^T, respectively.

If we assume that processor k stores matrices A_{kk}, A_{km} and A_{mk} in its local memory, it is obvious that it can compute L_{kk}, U_{kk}, U_{km} and L_{mk} *independently*. As a result, all the blocks of L and U except for L_{mm} and U_{mm} can be obtained in parallel. To see how these last two matrices can be determined, we should observe that processor k can also form the product $L_{mk}U_{km}$, and send it to processor m. If these terms are added along the way (using an algorithm similar to the one that we discussed in the context of scalar products), processor m will receive matrix

$$S_m = \sum_{k=1}^{m-1} L_{mk}U_{km} \tag{70}$$

This matrix is needed in the final step of the factorization, because equation (64) indicates that A_{mm} can be expressed as

$$A_{mm} = L_{mm}U_{mm} + S_m \tag{71}$$

As a result, processor m can compute L_{mm} and U_{mm} by factorizing matrix $\tilde{A}_{mm} = A_{mm} - S_m$.

This type of parallelism has some major advantages, because it allows us to specify the computational tasks that each processor will perfom *ahead of time*. It also minimizes the communication overhead, since processors have to exchange information only in the final step of the factorization. In that respect, BBD decompositions have a decided advantage over the minimal degree ordering.

In the remainder of this section, we will consider two techniques that are capable of producing BBD structures. It should be noted that each of them can be combined with the minimal degree ordering, in the sense that the diagonal blocks can be internally permuted to minimize fill-in.

Nested Dissection

The nested dissection method is one of the first algorithms that was developed for permuting sparse matrices into the BBD form. It is simple to implement and tends to work well for matrices that exhibit some form of regularity in their nonzero pattern. We will see, however, that it is not as effective when matrix A has an irregular structure (which is typical for electric circuits, for example).

In order to describe how this algorithm works, we first need to introduce several definitions from graph theory.

Definition 1. The *eccentricity* of node x in a graph (which is denoted by $l(x)$) represents the maximal distance between x and any other node in the graph.

Definition 2. The *diameter* of graph G (denoted $\delta(G)$) is the maximal distance between any two nodes in G. In light of Definition 1, this quantity can be described as

$$\delta(G) = \max_{x \in G} l(x) \tag{72}$$

Definition 3. Node x is said to be a *peripheral node* in graph G if $l(x) = \delta(G)$.

Since finding peripheral nodes in a large graph can be a difficult task in general, the nested dissection algorithm focuses on a simpler problem, and searches instead for a node whose eccentricity is as large as possible. Such a node is said to be *pseudo-peripheral*, and the following example illustrates how it can be identified.

Example 7. Consider the 8×8 matrix

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & * & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & * & * \\ 0 & * & * & * & * & * & * & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & * & * & * & * & * \\ 0 & 0 & 0 & * & 0 & 0 & * & * \end{bmatrix} \tag{73}$$

whose graph-theoretic representation is shown in Fig. 22.

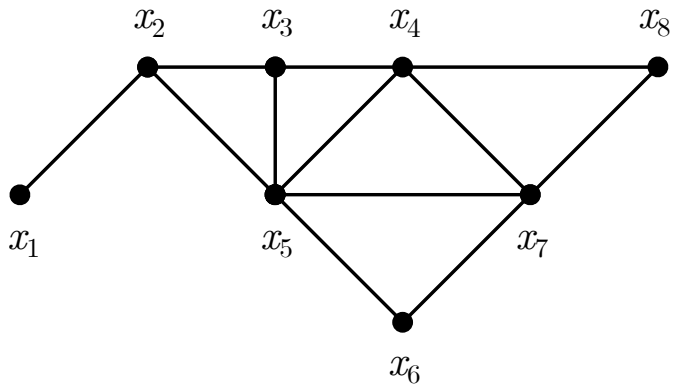

Figure 22: The graph that corresponds to the matrix in (73).

We will start by randomly choosing a node x_R, and grouping all the other nodes in the graph into different levels, based on their distance from x_R. If we choose $x_R = x_3$, for example, we obtain the structure shown in Fig. 23 (which is commonly referred to as a *rooted level structure*). In this diagram, the lines that connect different levels represent a subset of edges from the original graph.

If we take a closer look at Fig. 23, we will observe that x_3 is not a good candidate for a pseudo-peripheral node, since its distance to any other node in the graph is no larger than 2. We therefore need to perform another iteration, starting from the node in the last level which has the *lowest degree*. In our example, this happens to be x_1, and the level structure that is rooted at this node is shown in Fig. 24.

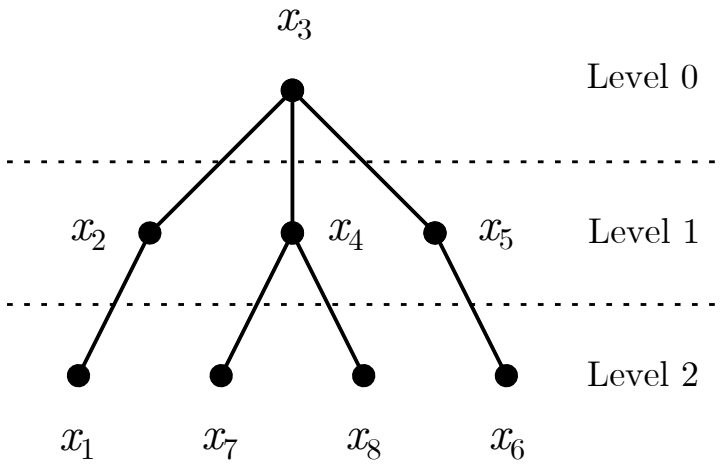

Figure 23: The level structure rooted at x_3.

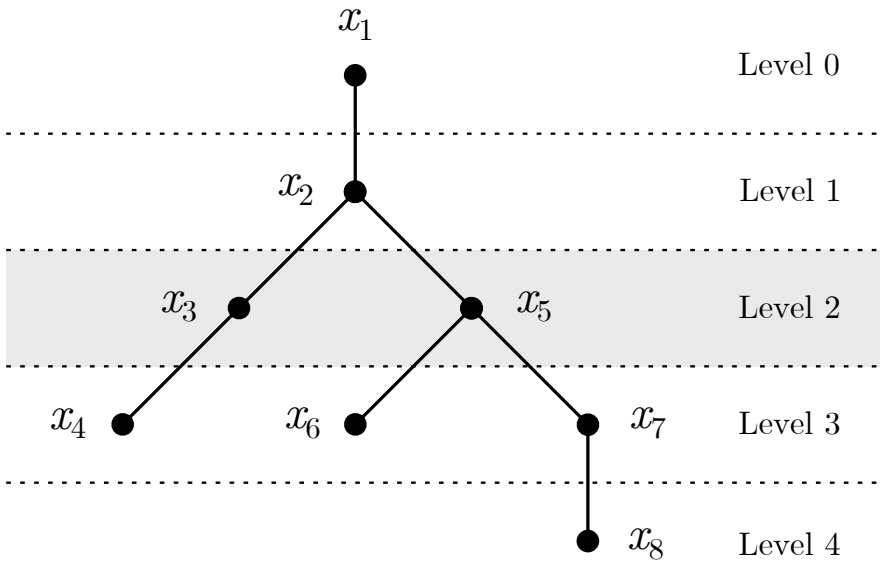

Figure 24: The level structure rooted at x_1.

In order to optimize the BBD structure, this process should continue until the number of levels stops increasing. It is easily verified that this occurs when we form a level structure rooted at node x_8 (since it has as many levels as the one that is rooted at node x_1). At this point, we can claim that we have found a pseudo-peripheral node, which could be either x_1 or x_8.

Once such a node is identified, the nested dissection algorithm proposes that we remove the *middle level* of the structure, together with all the links that connect it to other levels (this level is shaded gray in Fig. 24). When we do so, we obtain two disconnected subgraphs (which consist of nodes $\{x_1,\ x_2\}$ and $\{x_4,\ x_6, x_7, x_8\}$, respectively), and a *separator* whose elements are nodes $\{x_3,\ x_5\}$. This tells us that a permutation defined by vector

$$p = \begin{bmatrix} 1 & 2 & 4 & 6 & 7 & 8 & 3 & 5 \end{bmatrix} \tag{74}$$

will produce a BBD matrix whose structure is

$$\tilde{A} = \left[\begin{array}{cc|cccc|cc} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & 0 & 0 & 0 & 0 & * & * \\ \hline 0 & 0 & * & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & 0 & 0 & * \\ 0 & 0 & * & * & * & * & 0 & * \\ 0 & 0 & * & 0 & * & * & 0 & 0 \\ \hline 0 & * & * & 0 & 0 & 0 & * & * \\ 0 & * & * & * & * & 0 & * & * \end{array}\right] \tag{75}$$

In this matrix, the separator obviously corresponds to the border.

Example 7 suggests that the nested dissection algorithm doesn't necessarily produce diagonal blocks of equal (or even similar) sizes. This is clearly undesirable from the standpoint of parallel computing, since it could result in poor load balancing (particularly if the discrepancy is significant). We should also bear in mind that this decomposition can be applied recursively to the diagonal blocks, which results in a nested structure such as the one shown in Fig. 25. This type of ordering is suitable when the matrix is large, but the parallelization is somewhat more complicated to implement, since it requires a hierarchical distribution of tasks across the processors.

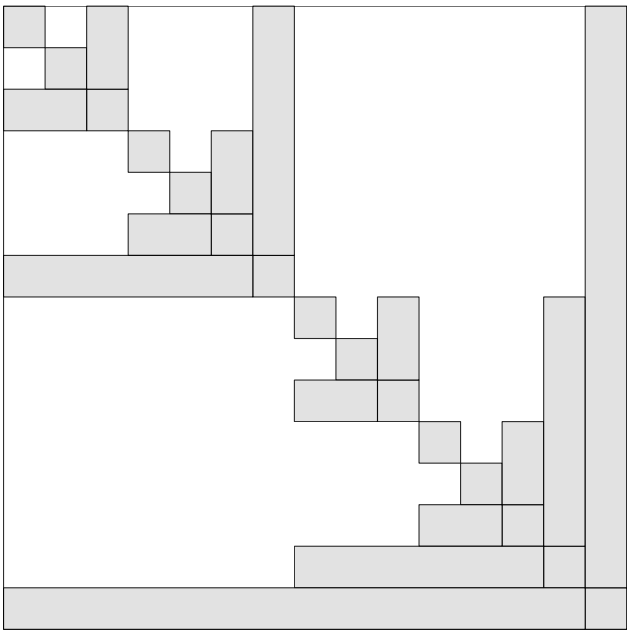

Figure 25: A nested BBD structure.

As noted earlier, we can reduce fill-in in BBD structures by permuting each diagonal block internally (typically, using the minimal degree method). This is not so easy to do with last diagonal block, however, because matrix $\tilde{A}_{mm} = A_{mm} - S_m$ is generally *not* sparse (due to the way S_m is formed). One way to deal with this problem is to minimize the size of the

border as much as possible, since that would reduce the dimensions of matrix $\tilde{A}_{mm}$. If we manage to make this matrix sufficiently small, its factorization won't be too time consuming even if all of its elements had nonzero values.

The nested dissection algorithm can minimize the border quite effectively for orderly configurations, but it doesn't work as well for other types of matrices. The following example amplifies this point, and illustrates the limitations of this approach.

Example 8. Consider the graph shown in Fig. 26, which corresponds to a sparse matrix of dimension 28×28. The level structure that is produced by selecting node x_1 as the root is shown in Fig. 27.

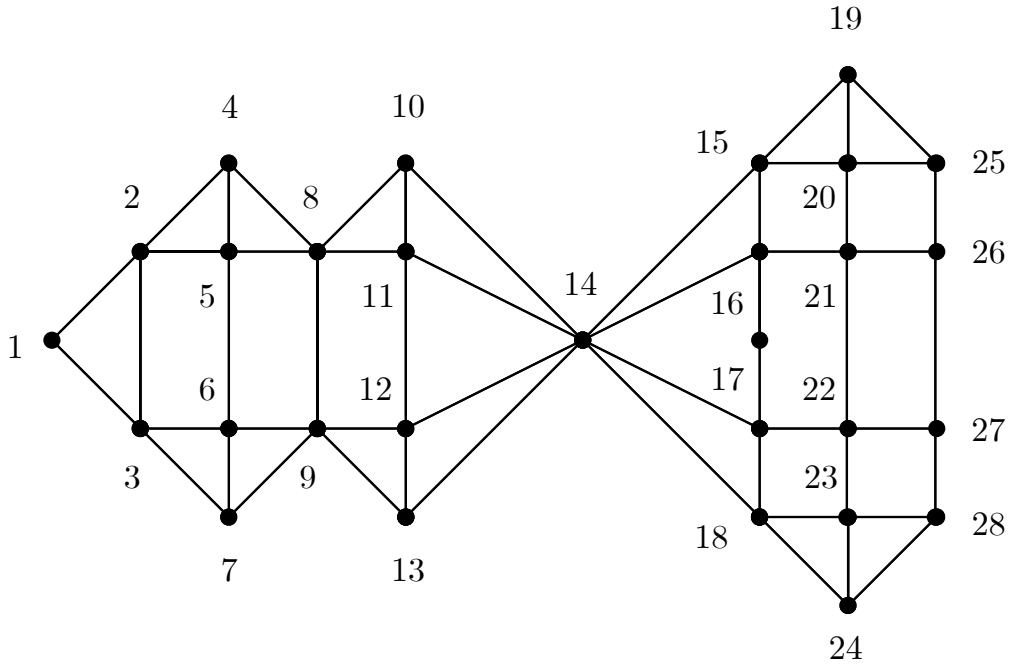

Figure 26: Graph of an unstructured sparse matrix.

If we follow the nested dissection algorithm to the letter and remove the middle level from this structure, we will obtain two diagonal blocks of dimensions 9×9 and 14×14, respectively, and a border block of size 4×4 (which corresponds to vertices $\{x_{10}, x_{11}, x_{12}, x_{13}\}$). It is readily observed, however, that we could get a better BBD structure by removing only vertex x_{14}, in which case we would have diagonal blocks of dimensions 13×13 and 14×14, and a border block of size 1×1. This would clearly result in a more efficient parallelization, but the nested dissection algorithm would not be able to produce such a decomposition.

Spectral Partitioning Methods

Spectral partitioning methods are based on the eigenvalues and eigenvectors of matrices that are used to represent undirected graphs. One of the most commonly used matrices for this purpose is the so-called *Laplacian* of graph G, whose description relies on the following definition.

Definition 4. The *adjacency matrix* M that is associated with graph G has zeros on its diagonal, and its off-diagonal entries satisfy

$$m_{ij} = \begin{cases} 1 & \text{if vertices } x_i \text{ and } x_j \text{ are connected} \\ 0 & \text{if vertices } x_i \text{ and } x_j \text{ are } \textit{not} \text{ connected} \end{cases} \tag{76}$$

The Laplacian of G (which we will denote by Q) is related to the incidence matrix as

$$Q = D - M \tag{77}$$

where D is a diagonal matrix in which d_{ii} represents the *degree* of node i.

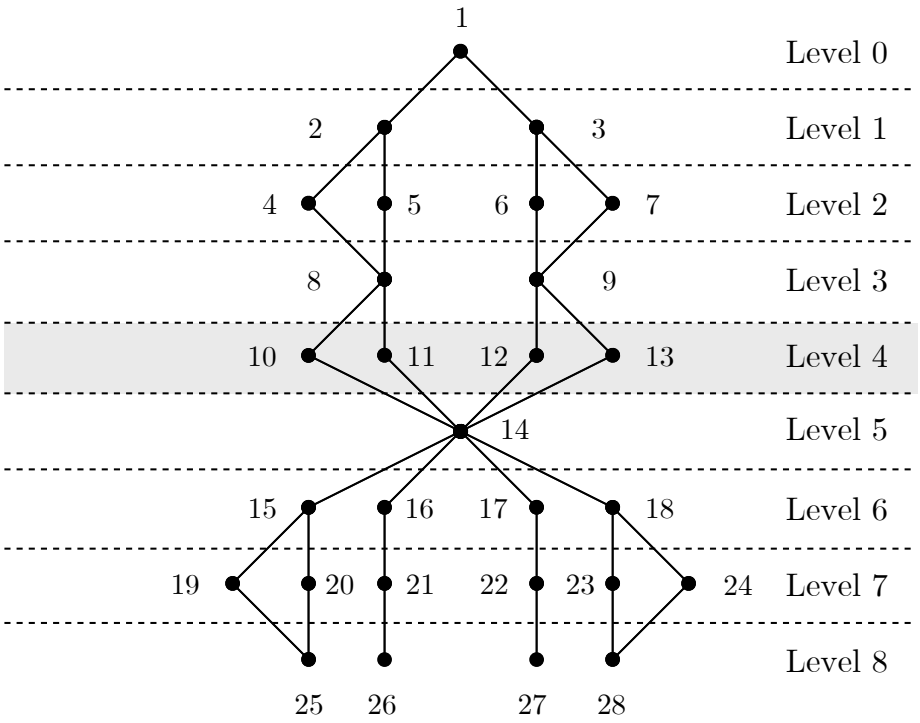

Figure 27: The level structure rooted at x_1.

Since matrix Q is symmetric by construction, its eigenvalues must be real (see Theorem 1.1 for a proof of this property). It can also be shown that all of these eigenvalues satisfy $\lambda_i \geq 0$, and that at least one of them equals zero. With that in mind, in the following we will denote the smallest *positive* eigenvalue of Q by λ_m and the corresponding eigenvector by X_m. Using this notation, the decomposition algorithm can be described as a sequence of four steps.

STEP 1. Compute eigenvector $X_m = [x_1\ x_2 \ldots x_n]^T$, and determine its *median component* (which we will denote by x_l).

STEP 2. Partition the vertices of graph G into two sets, A and B, so that vertex $i \in A$ if $x_i > x_l$ and $i \in B$ otherwise. Given that x_l is the median component of X_m, this ensures that sets A and B will have approximately the same size.

STEP 3. Find the minimal set of vertices in the graph whose removal will eliminate all edges that connect sets A and B. This set represents the *separator*.

STEP 4. Once the separator is removed, repeat Steps 1 - 3 on the remaining components as many times as necessary.

One of the advantages of this algorithm is that it guarantees similar block sizes, which ensures good load balancing. This approach also produces small borders, since it is designed to minimize the size of the separator. The following example further illustrates this point, and demonstrates how the algorithm works in practice.

Example 9. Consider a 9×9 matrix whose nonzero pattern is

$$A = \begin{bmatrix} * & 0 & * & 0 & 0 & 0 & * & 0 & 0 \\ 0 & * & 0 & * & * & * & * & 0 & 0 \\ * & 0 & * & 0 & 0 & 0 & * & 0 & 0 \\ 0 & * & 0 & * & * & 0 & 0 & 0 & * \\ 0 & * & 0 & * & * & 0 & 0 & 0 & * \\ 0 & * & 0 & 0 & 0 & * & 0 & 0 & * \\ * & * & * & 0 & 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 & * \end{bmatrix} \tag{78}$$

The graph that corresponds to this matrix is shown in Fig. 28.

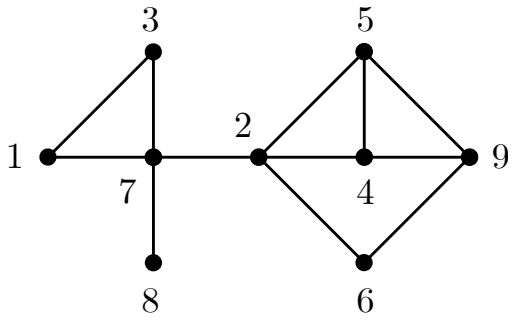

Figure 28: Graph that corresponds to the matrix in (78).

From this graph, we easily obtain the Laplacian matrix as

$$Q = D - M = \begin{bmatrix} 2 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 4 & 0 & -1 & -1 & -1 & -1 & 0 & 0 \\ -1 & 0 & 2 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 3 & -1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & -1 & 3 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 0 & 2 & 0 & 0 & -1 \\ -1 & -1 & -1 & 0 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 3 \end{bmatrix} \tag{79}$$

The smallest nonzero eigenvalue of Q is $\lambda_m = 0.30969$, and the corresponding eigenvector is

$$X_m = \begin{bmatrix} 4.0 & -1.8 & 4.0 & -3.1 & -3.1 & -3.1 & 2.8 & 4.0 & -3.5 \end{bmatrix}^T \tag{80}$$

(for the sake of simplicty, the components of X_m have been rounded off to a single decimal).

It is not difficult to see that the median component of X_m is $x_l = -1.8$. According to the algorithm, sets A and B will then be $A = \{x_1, x_3, x_7, x_8\}$ and $B = \{x_2, x_4, x_5, x_6, x_9\}$, respectively. If we now group the nodes in the manner shown in Fig. 29 (which corresponds to the partitioning defined by sets A and B), it becomes apparent that removing edge $(2, 7)$ will result in two disconnected graphs.

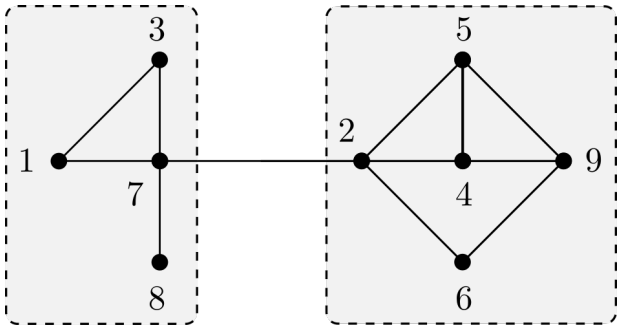

Figure 29: The partitioned graph.

The simplest way to accomplish this is to eliminate node x_2 (and all its incident edges). This means that x_2 should be the separator, and that the permutation vector should be

$$p = \begin{bmatrix} 1 & 3 & 7 & 8 & 4 & 5 & 6 & 9 & 2 \end{bmatrix} \tag{81}$$

When we apply this permutation to matrix A, we obtain a BBD structure of the form

$$\tilde{A} = \left[\begin{array}{cccc|cccc|c} * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 & * \\ 0 & 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & * & * & 0 & * & * \\ 0 & 0 & 0 & 0 & * & * & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & * & * & * & * & 0 \\ \hline 0 & 0 & * & 0 & * & * & * & 0 & * \end{array}\right] \tag{82}$$

Note that the diagonal blocks are balanced in size (both have dimension 4×4), and that the border is as small as it can possibly be.

Remark 1. When analyzing the relative merits of this algorithm, it is important to keep in mind that calculating eigenvectors and eigenvalues can pose some major challenges when the matrix is large. It is therefore worth exploring alternative strategies that require less computation (and are therefore easier to implement). One such possibility is described in the textbook.

Epsilon Decomposition

Epsilon decomposition is a technique that is not limited to sparse matrices (or structurally symmetric matrices, for that matter). As such, it is more general, and can be used in cases when the other methods that we discussed so far do not apply. The basic idea behind this approach is very simple - all we need to do is choose a positive number ε, and then eliminate all elements in the matrix whose magnitude is *no larger* than this value. Since this inevitable "sparsifies" the original matrix, there is a good chance that it can be permuted into a block diagonal form for certain values of ε.

To see why such a decomposition is desirable when it comes to solving system

$$Ax = b \tag{83}$$

iteratively, we should first recall that this process can be described as

$$x(k+1) = Gx(k) + \xi \tag{84}$$

where

$$G = I - Q^{-1}A \tag{85}$$

and

$$\xi = Q^{-1}b \tag{86}$$

We previously established that sequence $\{x(k)\}$ will converge to the solution of equation (83) if $\rho(G) < 1$. Because $\rho(G) \leq \|G\|$ for *any* choice of norm, convergence can also be guaranteed whenever $\|G\| < 1$ (see Lemma 10.1 for a proof of this property).

Epsilon decomposition can help us satisfy this condition because it allows us to express matrix A as

$$A = A_D + \varepsilon A_C \tag{87}$$

where A_D is block diagonal, and all the elements of A_C have magnitudes that are no larger than 1. If we choose matrix Q as

$$Q = A_D \tag{88}$$

we obtain

$$G = I - Q^{-1}A = I - A_D^{-1}(A_D + \varepsilon A_C) = \varepsilon A_D^{-1} A_C \tag{89}$$

Given that

$$\|G\| = \varepsilon \left\| A_D^{-1} A_C \right\| \tag{90}$$

it is readily observed that G will satisfy $\|G\| < 1$ when ε is sufficiently small. This will not only ensure convergence, but can also significantly accelerate it in some cases.

The following example illustrates how epsilon decomposition works in practice.

Example 10. Consider the matrix

$$A = \begin{bmatrix} 1 & 0 & 0.7 & 0 & 0 & 0 \\ 0 & 1 & 0.3 & 0 & 0 & 0 \\ 0.1 & 0 & 1 & 0.1 & 0.25 & 0 \\ 0 & 0 & 0 & 1 & 0.1 & 1 \\ 0.1 & 0 & 1 & 0 & 1 & 0.1 \\ 0 & 0.1 & 0 & 0.25 & 0 & 1 \end{bmatrix} \tag{91}$$

which is clearly *not* symmetric. With this matrix we can associate a bipartite graph like the one shown in Fig. 30, where x_i refers to column i and y_j to row j. The procedure for forming such a graph is quite straightforward - given a column x_k, match it to the corresponding row y_k, and then connect y_k to all other columns x_i for which $a_{ki} \neq 0$. We normally start the procedure from node x_1, and end it when all the rows and columns have appeared in the graph.

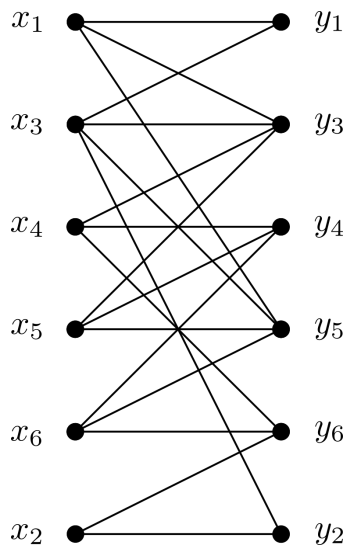

Figure 30: Bipartite graph that corresponds to the matrix in (91).

The diagram in Fig. 30 indicates that matrix A cannot be permuted into a block diagonal form, since the corresponding bipartite graph does not consist of two or more disconnected components. To see how epsilon decomposition can help produce such components, suppose that we pick $\varepsilon = 0.1$. Doing so will obviously eliminate all elements whose magnitude is 0.1 or smaller, which produces a "sparsified" matrix

$$A_\varepsilon = \begin{bmatrix} 1 & 0 & 0.7 & 0 & 0 & 0 \\ 0 & 1 & 0.3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.25 & 0 & 1 \end{bmatrix} \tag{92}$$

If we now represent matrix A_ε as a bipartite graph (following the procedure that we just described), we will obtain the structure shown in Fig. 31. It is important to recognize at this point that nodes x_3 and x_5 appear *more than once* in this figure. Whenever that happens, we need to merge the blocks that have common nodes (in order to avoid redundancy). If we do so, we obtain the graph in Fig. 32, which consists of two disconnected components.

The ordering of the nodes in this figure tells us that matrix A_ε can be transformed into a block diagonal form using permutation vector

$$p = \begin{bmatrix} 1 & 2 & 3 & 5 & 4 & 6 \end{bmatrix} \tag{93}$$

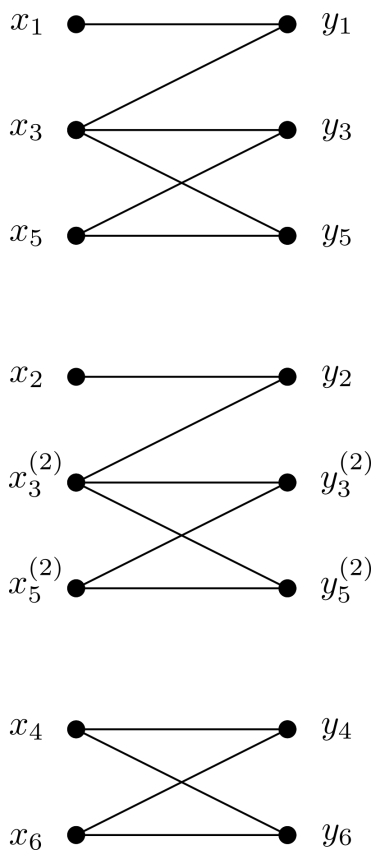

Figure 31: Decomposition with overlapping for $\varepsilon = 0.1$.

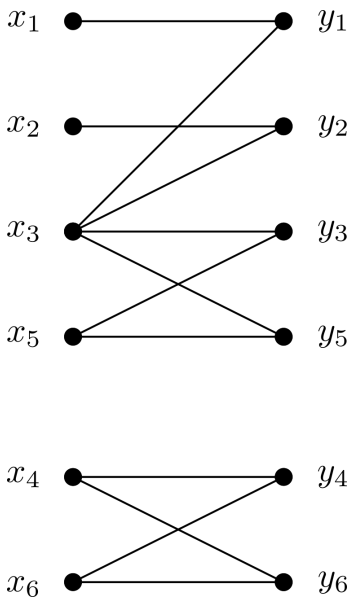

Figure 32: Decomposition when overlapping components are aggregated.

If we apply this permutation to matrix A, the resulting matrix $\tilde{A}$ can be decomposed as

$$\tilde{A} = \tilde{A}_D + \varepsilon \tilde{A}_C \tag{94}$$

where

$$\tilde{A}_D = \left[\begin{array}{cccc|cc} 1 & 0 & 0.7 & 0 & 0 & 0 \\ 0 & 1 & 0.3 & 0 & 0 & 0 \\ 0.1 & 0 & 1 & 0.25 & 0 & 0 \\ 0.1 & 0 & 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0.25 & 1 \end{array}\right] \tag{95}$$

and

$$\tilde{A}_C = \left[\begin{array}{cccc|cc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right] \tag{96}$$

Such a structure is not ideal for parallel computing (because the block sizes are not balanced), but it is clearly better than the one we started with, so we should continue looking for an appropriate choice of ε.

If we now increase ε to 0.3, we obtain matrix

$$A_\varepsilon = \begin{bmatrix} 1 & 0 & 0.7 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{97}$$

Applying the epsilon decomposition to A_ε produces the bipartite graph shown in Fig. 33, which assumes the form shown in Fig. 34 after blocks 1 and 4 are merged (since they have common elements).

From Fig. 34, we obtain permutation vector

$$p = \begin{bmatrix} 1 & 3 & 5 & 2 & 4 & 6 \end{bmatrix} \tag{98}$$

which transforms the original matrix A into

$$\tilde{A} = \tilde{A}_D + \varepsilon \tilde{A}_C \tag{99}$$

where

$$\tilde{A}_D = \left[\begin{array}{ccc|ccc} 1 & 0.7 & 0 & 0 & 0 & 0 \\ 0.1 & 1 & 0.25 & 0 & 0 & 0 \\ 0.1 & 1 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0.1 & 0.25 & 1 \end{array}\right] \tag{100}$$

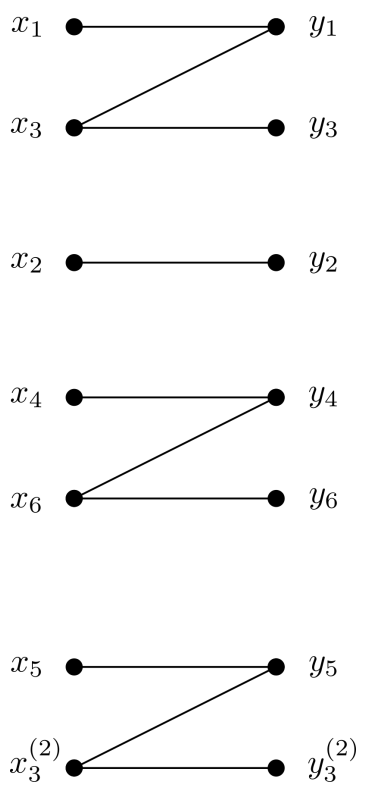

Figure 33: Decomposition with overlapping for $\varepsilon = 0.3$.

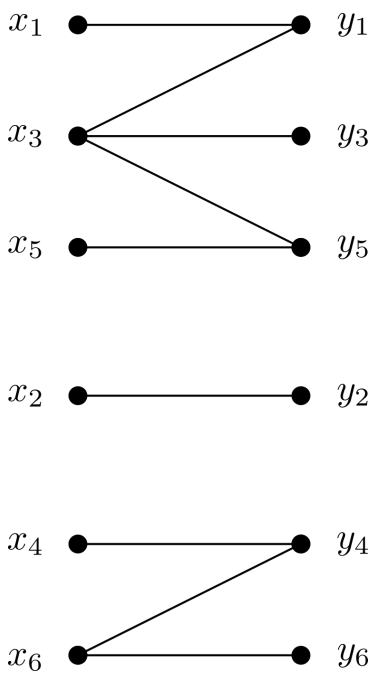

Figure 34: Decomposition when overlapping components are aggregated.

and

$$\tilde{A}_C = \left[\begin{array}{ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.33 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.33 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.33 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \tag{101}$$

Remark 2. Note that we grouped node x_2 with nodes x_4 and x_6, in order to obtain two diagonal blocks of equal sizes. By doing so, we produced a structure that is well suited for parallel computation. This comes at the expense of increasing ε, however, and will require more iterations than the case where $\varepsilon = 0.1$.

Remark 3. The tradeoff between block sizes and the convergence rate is something that needs to be monitored carefully when applying this technique, because increasing ε too much can slow the iterative process down to a point where it becomes inefficient. On the other hand, choosing a very small ε usually fails to produce a sufficient number of diagonal blocks, which limits our ability to parallelize the computations.

As a final point in this topic, we should note that epsilon decomposition can be uesful even if matrix A has only a handful of small elements. The following example illustrates how scaling can help in such cases.

Example 11. Consider system

$$Ax = b \tag{102}$$

where

$$A = \begin{bmatrix} 5 & 2 & 0 & 1 \\ 0.1 & 0.1 & 0.01 & 0 \\ 2 & 2 & 8 & 4 \\ 2 & 2 & 6 & 10 \end{bmatrix} \tag{103}$$

and

$$b = \begin{bmatrix} 5 \\ 0 \\ 2 \\ 5 \end{bmatrix} \tag{104}$$

If we were to set $\varepsilon = 0.1$, matrix A_ε would have the form

$$A_\varepsilon = \begin{bmatrix} 5 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 2 & 2 & 8 & 4 \\ 2 & 2 & 6 & 10 \end{bmatrix} \tag{105}$$

which is clearly not suitable for a decomposition. Since there are no other reasonable choices for ε in this case, it would appear that there is nothing more that we can do.

It turns out, however, that there is a simple way to resolve this problem. To see how that can be done, suppose that we multiply both sides of (102) by matrix

$$D = \begin{bmatrix} 0.2 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 0.125 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix} \tag{106}$$

whose elements are reciprocals of the diagonal elements in A. If we do so, the system will take the form

$$\bar{A}x = \bar{b} \tag{107}$$

where

$$\bar{A} = \begin{bmatrix} 1 & 0.4 & 0 & 0.2 \\ 1 & 1 & 0.1 & 0 \\ 0.25 & 0.25 & 1 & 0.5 \\ 0.2 & 0.2 & 0.6 & 1 \end{bmatrix} \tag{108}$$

and

$$\bar{b} = \begin{bmatrix} 1 \\ 0 \\ 0.25 \\ 0.5 \end{bmatrix} \tag{109}$$

Matrix $\bar{A}$ can now obviously be expressed as

$$\bar{A} = \bar{A}_D + \varepsilon \bar{A}_C \tag{110}$$

where

$$\bar{A}_D = \left[\begin{array}{cc|cc} 1 & 0.4 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0.6 & 1 \end{array}\right] \tag{111}$$

$$\bar{A}_C = \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0.4 & 0 \\ \hline 1 & 1 & 0 & 0 \\ 0.8 & 0.8 & 0 & 0 \end{array}\right] \tag{112}$$

and $\varepsilon = 0.25$. As a result, equation (102) can be solved iteratively using Jacobi's method.

Remark 4. In this example we did not use bipartite graphs, because matrix $\bar{A}$ already has the desired structure. In general, this will not be the case, and it is usually necessary to perform a permutation in order to produce a matrix of the form (110).