An Introduction to Direct Sparse Solvers

Sam Blackwood & Ahyo Falick

December 2021

Abstract

We discuss two direct methods for solving a linear system when the corresponding matrix is sparse. We start with an algorithm that applies to tridiagonal matrices (a subclass of sparse matrices) which requires $\mathcal{O}(n)$ operations to solve. The second method we consider applies to any sparse matrix. This technique involves creating a graph structure from the given matrix and recursively partitioning the graph in such a way that improves the sparse structure of the matrix. The method, known as nested dissection, allows for parallel processing and has a time complexity around $\mathcal{O}(n^{3/2})$ (can vary based on the implementation) to reorder the matrix and consequently solve the system. We then supply several numerical examples to illustrate the performance of these two techniques. We conclude with a discussion of possible future work in sparse solvers.

1 Introduction

In many applications of science and engineering, situations arise where one has to solve the system $\mathrm{A}\mathbf{x} = \mathbf{b}$, where A is a sparse matrix. There is no formal definition of a sparse matrix, but it is often convenient to consider a matrix sparse if it has $\mathcal{O}(n)$ nonzero elements, where A is $n \times n$. If one were to attempt to solve $\mathrm{A}\mathbf{x} = \mathbf{b}$ naively with Gaussian elimination[1], the solve would take $\mathcal{O}(n^3)$ operations. In practice, matrices are often large in size, and thus obtaining the solution in n^3 time is unfeasible. Fortunately, methods exist that exploit the sparse structure of these matrices and can significantly improve the performance of solving the system. These methods are called sparse solvers. In this paper, we will investigate the Thomas algorithm, which is a sparse solver used in the case of a tridiagonal matrix, and nested dissection, which can be applied to any sparse matrix. But before we can properly address these algorithms, it is import to understand where the need for sparse solvers comes from.

The history of sparse solvers dates back to the late 1950s when researchers in linear programming and electrical power systems analysis began to solve real-world problems on computers. They noticed that many of these problems were defined by sparse matrices, often with a well-defined pattern. Given that computers at that time had very limited memory and slow processors, the researchers began to realize that to solve such systems they would have to exploit the sparsity patterns. This led to the creation of sparse matrix research. Within this field, there are two distinct areas: (1) research in reducing the time to solve a sparse system and (2) research in reducing the storage space of sparse matrices. Our paper will primarily focus on the first area of research, but it should be noted there has also been extensive research in improving the data structures of sparse matrices to reduce their space complexity.

To get a sense of just how prevalent sparse systems are in science and engineering, we supply a list of many of the applications of sparse matrices. This list is given by Derek O'Connor in his book, *An Introduction to Sparse Matrices*, and reads as follows:

> Sparse matrices occur naturally in the solution of many practical problems, e.g. electrical, gas; and water distribution systems; civil and mechanical engineering (structural analysis); production

[1]For the purposes of this paper, Gaussian elimination will refer to the process of creating an augmented matrix from the initial system, performing row operations to reduce the matrix to row echeclon form, and then performing back substitution to find the solution.

and financial planning (inventory control and portfolio selection); national and local government operations (income tax analysis and scheduling of fire and ambulance services); economics (input-output analysis). At a more theoretical level sparse matrices arise in Graph Theory, Linear Programming, Finite Element Methods, and the solution of ordinary and partial differential equations.

As one can see, sparse matrices are omnipresent in our world. It is no wonder that so much time and effort has been spent in developing more efficient methods for storing and solving such systems.

With regard to the first sparse solver discussed in this paper, the Thomas algorithm essentially improves upon Gaussian elimination for a tridiagonal system. In regular Gaussian elimination, the algorithm would zero out each entry below the entry a_{ii} for $i = 1, ..., n-1$ in order to create an upper triangular matrix. However, most of this work is unnecessary as all but one of the entries below the diagonal in a tridiagonal matrix is 0 by definition. The Thomas algorithm uses this fact as the basis of its derivation. Section 3.1 discusses this notion in more detail.

As for nested dissection, this method is centered around creating a graph structure out of the given matrix. The key idea associated with nested dissection is that the connections within the graph correspond to dependencies within the matrix. Therefore, if we can partition the graph into two disjoint subgraphs, then we can process each subgraph in parallel without having to worry about the solution of the first subgraph being dependent on the solution of the second subgraph (as disjoint implies no dependency between subgraphs one and two). It follows that this can be done recursively to produce even smaller subgraphs each of which becomes less expensive to process. Throughout this recursive process, nested dissection is creating a fill-reducing ordering of the initial matrix that allows us to best preserve the sparsity of the initial matrix upon performing an LU factorization. This is the general idea of nested dissection. We discuss this method in greater detail in section 3.2.

We have organized the rest of the paper as follows. Section 2 contains a review of topics in linear algebra and graph theory which aid in the understanding of sparse solvers. The bulk of our work is located in section 3 where we explore the two sparse solvers of interest in this paper. Section 4 presents our numerical findings associated with those methods. In section 5, we extend our study of nested dissection to runtime analysis. Section 6 summarizes our work and touches on future exploration within the context of direct sparse solvers.

2 Framework for Sparse Solvers

The foundation of sparse solvers relies on techniques found in linear algebra as well as graph theory. Therefore, we have deemed it appropriate to supply a brief review of certain topics in both fields. We start by supplying a list of definitions of terms and phrases to which we will frequently refer throughout the paper.

2.1 Definitions

Definition 2.1.1 (Symmetric matrix) *In this paper, we will define a symmetric matrix to be one where for every nonzero entry a_{ij}, there is a corresponding nonzero (not necessarily equivalent in value) entry a_{ji}. Essentially, a matrix is symmetric if it is symmetric in structure, not in numerical value, which is the traditional definition of a symmetric matrix. Any matrix that does not fit this definition is classified as* ***unsymmetric****.*

Definition 2.1.2 (nnz(s)) *Short for number of nonzero(s).*

Definition 2.1.3 (Fill-in) *Refers to the creation of nonzero entries in places that were initially zero, specifically when performing Gaussian Elimination/LU decomposition.*

Definition 2.1.4 (Fill-reducing ordering) *A fill-reducing ordering is a reordering of a matrix so that when an LU decomposition is performed on the matrix, the resulting L and U matrices have a reduced amount of*

fill-in compared to the initial L and U matrices produced from an LU on the initial matrix. For simplicity, we will also refer to this simply as the ***reordering****.*

Definition 2.1.5 (Partitioning) *Refers to the dividing of a graph into distinct parts using an edge or vertex separator. Also referred to as* ***separating****. See section* ***2.4*** *for more details.*

Definition 2.1.6 (Factorization) *Describes the method of decomposing a matrix into a product of two matrices using linear algebra techniques. These methods are discussed further in section* ***2.3****. Also referred to as* ***decomposition****.*

Definition 2.1.7 (Planar Graph) *A planar graph is a graph that can be drawn in the plane so that no edges cross.*

2.2 Gaussian Elimination

Gaussian elimination was one of the first algorithms to ever describe a process for solving an $n \times n$ system of equations. This algorithm is still taught to students learning linear algebra for the first time as it is easy to grasp and reasonable to use in cases that n is small. However, Gaussian elimination has numerous practicality issues. In the case of a sparse matrix, Gaussian elimination does much more work than is needed ($\mathcal{O}(n^3)$ operations). This is where the need for sparse solvers comes from. Another issue is more general and comes from the fact that it is common to want to solve $A\mathbf{x} = \mathbf{b}_j$ for an arbitrary number of $\mathbf{b}_j$ vectors. If we were to solve using the matrix A, we would have to perform Gaussian elimination for each $\mathbf{b}_j$ vector. However, if we were able to factor A in such a way that allows us to solve the system without repeating Gaussian elimination each time, then we could improve the performance of all the solves after the initial factorization. This is the motivation for our next discussion on matrix factorization.

2.3 Matrix Factorization

Whenever you call a linear solve on your machine, it is almost guaranteed that the software is performing some type of matrix factorization in order to compute the solution. Therefore, it is a topic worth discussing to better understand sparse solvers. The LU factorization involves breaking a given matrix A into two parts: a lower triangular matrix with ones along the diagonal (i.e. unitriangular) and an upper triangular matrix. The matrix U is obtained from Gaussian elimination (i.e. it is the upper triangular matrix that is formed upon zeroing out the lower diagonal of the matrix). The matrix L is formed by keeping track of row operations performed when constructing the U matrix. The system now becomes $LU\mathbf{x} = \mathbf{b}$. Once we have obtained matrices L and U, we perform two linear solves:

$$\textbf{(1)} \text{ let } U\mathbf{x} = \mathbf{y} \text{ and solve } L\mathbf{y} = \mathbf{b}$$
$$\textbf{(2)} \text{ solve } U\mathbf{x} = \mathbf{y} \text{ for } \mathbf{x}.$$

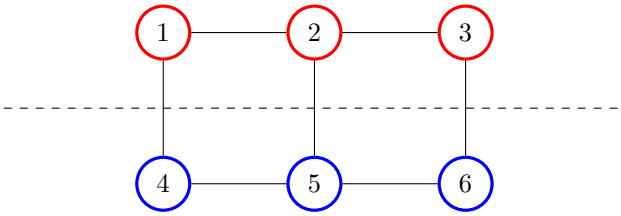
Since factoring the matrix requires performing Gaussian elimination on the system and keeping track of the row operations, the factorization is still $\mathcal{O}(n^3)$. But because both **(1)** and **(2)** are systems involving triangular matrices, the solve is now $\mathcal{O}(n^2)$ as opposed to $\mathcal{O}(n^3)$ in traditional Gaussian elimination. This is a significant enough of a difference to justify computing an LU decomposition before performing a linear solve on a machine.

2.4 Graph Separators

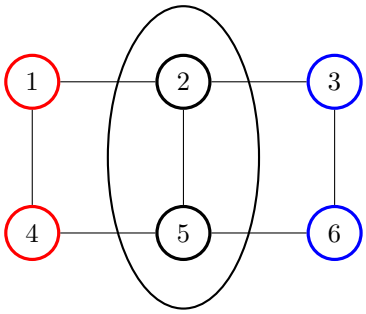
Graph separators have been greatly researched due to their wide range of use in applications. There are two main types of graph separators, edge separators and vertex separators.

Definition 2.4.1 (Edge separator) *Given a graph G = (V, E), an edge separator is a set of edges E' $\subset$ E that partitions (separates) V into two disjoint sets of vertices V_1 and V_2.*

Definition 2.4.2 (Vertex separator) *Given a graph G = (V, E), a vertex separator is a set of vertices V' $\subset$ V that partitions V into two disjoint sets of vertices V_1 and V_2.*



Edge separator with edges (1,4), (2,5), and (3,6) being the edges in E'



Vertex separator with vertices 2 and 5 in V'

Generally, with graph separators, we would like for $|V_1| \approx |V_2|$. As we will discuss later, nested dissection is concerned with finding a good (but not optimal) vertex separator within the given graph. Graph separators are important because they can be applied recursively to the partitioned subgraphs. In section 3.2 we will see how vertex separators are relevant to the problem of solving a sparse system, and what partitioning the graph actually means in the context of the initial matrix.

3 Sparse Solvers

As discussed earlier, sparse matrices appear in all sorts of practical applications. Taking advantage of the zero entries in sparse matrices can significantly reduce the number of floating point operations (flops) required, and thus improve computational efficiency. As the size of these matrices grow to support increasingly complex problems, efficiency and storage optimization become much more important topics of research. This section will analyze two of the methods used to efficiently solve these systems, namely the Thomas Algorithm and Nested Dissection. While we will provide our own Python code for the Thomas Algorithm, it should be noted that Nested Dissection is far more complicated to simulate and will therefore by analyzed using prewritten sources.

3.1 Thomas Algorithm

The Thomas Algorithm is a subset of Gaussian elimination in the case of a tridiagonal matrix and consists of two phases. In the first phase, the original coefficients are redefined as to eliminate the bottom diagonal of the tridiagonal system. After reducing the matrix, the second phase uses backward substitution to solve for all unknowns in the sparse linear system. The following exploration of the Thomas algorithm will include a more detailed discussion of its steps, derivation, and stability.

3.1.1 Definition of Algorithm

For a tridiagonal system of matrices defined as follows

$$\begin{bmatrix} a_1 & c_1 & 0 & ... & 0 \\ b_2 & a_2 & c_2 & & \vdots \\ 0 & b_3 & a_3 & \ddots & 0 \\ \vdots & & \ddots & \ddots & c_{n-1} \\ 0 & ... & 0 & b_n & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix}$$

the general solution using using the Thomas Algorithm [13] is:

1) Redefine coefficients as

$$c'_i = \begin{cases} \frac{c_i}{a_i} & i = 1 \\ \frac{c_i}{a_i - b_i c'_{i-1}} & i = 2, 3, ..., n-1 \end{cases} \text{ and } d'_i = \begin{cases} \frac{d_i}{a_i} & i = 1 \\ \frac{d_i - b_i d'_{i-1}}{a_i - b_i c'_{i-1}} & i = 2, 3, ..., n \end{cases}$$

2) Then, the solution to the linear system is found by back substitution

$$x_i = \begin{cases} d'_i & i = n \\ d'_i - c'_i x_{i+1} & i = n-1, n-2, ..., 1 \end{cases}$$

3.1.2 Derivation

To derive this algorithm, first let us rewrite the sparse matrix system as a sparse system of linear equations:

$$a_1 x_1 + c_1 x_2 = d_1 \tag{3.1.1}$$
$$b_2 x_1 + a_2 x_2 + c_2 x_3 = d_2 \tag{3.1.2}$$
$$b_3 x_2 + a_3 x_3 + c_3 x_4 = d_3 \tag{3.1.3}$$
$$\vdots \tag{3.1.4}$$
$$b_n x_{n-1} + a_n x_n = d_n \tag{3.1.5}$$

To simplify this matrix into an upperdiagonal one, we start by modifying the second equation using row arithmetic as follows:

$$(\text{Row } 2)a_1 - (\text{Row } 1)b_2$$

The second equation then becomes

$$(a_1 a_2 - b_2 c_1)x_2 + a_1 c_2 x_3 = a_1 d_2 - b_2 d_1$$

In the same way, we can redefine the coefficients for the third equation by using this newly defined second equation and the operation:

$$(\text{Row } 3)(a_1 a_2 - b_2 c_1) - (\text{Row } 2)b_3$$

This yields

$$(a_1 a_2 - b_2 c_1 - a_1 c_2 b_3)x_3 + (a_1 a_2 - b_2 c_1)c_3 x_4 = (a_1 a_2 - b_2 c_1)d_3 - b_3(a_1 d_2 - b_2 d_1)$$

We can continue in this way until all rows have been reduced because the final equation will have only one unknown. So we can use back substitution once we have solved for x_n. With the current technique, this will become quite complicated as can be seen from the first two iterations. To avoid this, we can find the pattern for coefficients and simply redefine them. These new coefficients are defined below.

$$a'_i = \begin{cases} a_i & i = 1 \\ a_i a'_{i-1} - c'_{i-1} b_i & i = 2, ..., n \end{cases} \quad c'_i = \begin{cases} c_i & i = 1 \\ c_i a'_i & i = 2, ..., n-1 \end{cases}$$

Note that b'_i is 0 for all $i = 2, ..., n$ because we are getting rid of the lower diagonal element for each iteration. The solution vector d' is also necessarily redefined to keep up with the row operations we've performed. The new solution vector has elements defined below

$$d'_i = \begin{cases} d_i & i = 1 \\ d_i a'_i - d_{i-1} b_i & i = 2, ..., n \end{cases}$$

This results in an upper triangular matrix, but we can further simplify the coefficients, by dividing by a'_i. Doing this results in the coefficients defined in the Thomas Algorithm definition above i.e

$$a'_i = 1 \text{ for all i}, \quad c'_i = \begin{cases} \frac{c_i}{a_i} & i = 1 \\ \frac{c_i}{a_i - b_i c'_{i-1}} & i = 2, 3, ..., n-1 \end{cases} \text{ and } d'_i = \begin{cases} \frac{d_i}{a_i} & i = 1 \\ \frac{d_i - b_i d'_{i-1}}{a_i - b_i c'_{i-1}} & i = 2, 3, ..., n \end{cases}$$

Then, we can form a new matrix system with the newly defined coefficients

$$\begin{bmatrix} 1 & c'_1 & 0 & ... & 0 \\ 0 & 1 & c'_2 & & \vdots \\ 0 & 0 & 1 & \ddots & 0 \\ \vdots & & \ddots & \ddots & c'_{n-1} \\ 0 & ... & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d'_1 \\ d'_2 \\ d'_3 \\ \vdots \\ d'_{n-1} \\ d'_n \end{bmatrix}$$

Now the only step left is to use backwards substitution to solve for the x vector. The last equation of the new system is now simply $x_n = d'_n$ and we can use this to solve for all other x values.

3.1.3 Example

To visualize the Thomas Algorithm at work, we present a simple example using an $n = 4$ tridiagonal linear system. First, let the equation $\mathrm{A}\mathbf{x} = \mathbf{d}$ be given by the matrix system

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 1 & 3 & -1 & 0 \\ 0 & 2 & 2 & 3 \\ 0 & 0 & 4 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 3 \\ -5 \end{bmatrix}$$

Now, we can create a vector of redefined coefficients for both the c-diagonal elements and the d-vector. So let

$$\vec{c'} = \begin{bmatrix} c'_1 \\ c'_2 \\ c'_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 3/4 \end{bmatrix}$$

And let

$$\vec{d'} = \begin{bmatrix} d'_1 \\ d'_2 \\ d'_3 \\ d'_4 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \\ 7/4 \\ 12/5 \end{bmatrix}$$

Now, we can easily find the solution to $\vec{x}$ using backwards substitution. In this way, we find that

$$\vec{x} = \begin{bmatrix} 61/10 \\ -41/20 \\ -1/20 \\ 12/5 \end{bmatrix}$$

3.1.4 Stability and Complexity

The Thomas Algorithm is generally stable for non-singular matrices except in rare cases. The condition for guaranteed stability is

$$||b_i|| > ||a_i|| + ||c_i||$$

for all i. Furthermore, this algorithm can be made stable through matrix pivoting techniques in the case that this condition is not met [12]. To determine the time complexity of the Thomas algorithm, we will count the number of flops required to compute the solution for n equations. First, we count the number of operations required to find the coefficients c'_i. Recall that

$$c'_i = \begin{cases} \frac{c_i}{a_i} & i = 1 \\ \frac{c_i}{a_i - b_i c'_{i-1}} & i = 2, 3, ..., n-1 \end{cases}$$

When $i = 1$, we only have 1 operation to perform (division). At each i for $i = 2, 3, ..., n-1$, we have to perform a division, a multiplication ($b_i c'_{i-1}$), and a subtraction ($a_i - b_i c'_{i-1}$), each of which is done $n-2$ times. Thus, for the c'_i coefficients, $1 + 3(n-2) = 3n-5$ operations are required. To determine how many flops are required to compute the d'_i coefficients, first recall

$$d'_i = \begin{cases} \frac{d_i}{a_i} & i = 1 \\ \frac{d_i - b_i d'_{i-1}}{a_i - b_i c'_{i-1}} & i = 2, 3, ..., n \end{cases}$$

When $i = 1$, once again, we only have to perform one division. At each i for $i = 2, 3, ..., n$, we have to compute one division, two multiplications, and two subtractions. However, note that for each i, one of the multiplication computations and one of the subtraction computations is the exact same as those in computing c'_i. Therefore, we can store these values and reuse them in computing d'_i. So, only three new operations are required for each $i = 2, 3, ..., n$. So the total operations taken to compute $d'_i = 1 + 3(n-1) = 3n-2$. The last bit of computation we need to consider is the back substitution phase. Recall that in solving for x_i, we have

$$x_i = \begin{cases} d'_i & i = n \\ d'_i - c'_i x_{i+1} & i = n-1, n-2, ..., 1 \end{cases}$$

When $i = n$, no operations are required. At each i for $i = n-1, n-2, ..., 1$, we have to compute one multiplication ($c'_i x_{i+1}$) and one subtraction ($d'_i - c'_i x_{i+1}$). So backsolving requires $2(n-1)$ flops. Thus, the total operation count for the Thomas algorithm is $3n-5+3n-2+2n-2 = 8n-9$ operations. Thus, the Thomas algorithm is a $\mathcal{O}(n)$ algorithm.

3.2 Nested Dissection

While the Thomas algorithm works nicely for tridiagonal linear systems, larger nontridiagonal sparse matrices require a more complex approach. One such method for solving these sparse systems is called Nested Dissection. This subsection will discuss a typical nested dissection algorithm and then provide a simple example to illustrate the technique.

3.2.1 Overview of Algorithm

Nested dissection is a sparse solving technique based on a divide and conquer heuristic that uses graph partitioning. There are, in fact, many similarities between sparse matrices and graph theory. The goal of this method is to find an ordering that minimizes fill-in when factorizing a sparse linear system. To do so requires smart and efficient reordering. As a simple example, consider a sparse matrix of the form

$$A = \begin{bmatrix} x & x & x & x & x \\ x & x & & & \\ x & & x & & \\ x & & & x & \\ x & & & & x \end{bmatrix}$$

This results in the following LU factorization:

$$LU = \begin{bmatrix} x & & & & \\ x & x & & & \\ x & x & x & & \\ x & x & x & x & \\ x & x & x & x & x \end{bmatrix} \begin{bmatrix} x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ & & & & x \end{bmatrix}$$

Remark that the initial matrix had the structure of an arrowhead pointing to the top left of the matrix. This is the least ideal structure to have when trying to avoid fill-in. However, if we were to reorder the matrix so that it takes the following structure

$$A = \begin{bmatrix} x & & & & x \\ & x & & & x \\ & & x & & x \\ & & & x & x \\ x & x & x & x & x \end{bmatrix}$$

It will result in a sparse LU factorization

$$LU = \begin{bmatrix} x & & & & \\ & x & & & \\ & & x & & \\ & & & x & \\ x & x & x & x & x \end{bmatrix} \begin{bmatrix} x & x & x & x & x \\ & x & & & \\ & & x & & \\ & & & x & \\ & & & & x \end{bmatrix}$$

This concept is one of the main motivators for nested dissection. It is important to note that nested dissection is a constantly expanding topic and thus has many unique approaches. For our discussion, we will focus on one of the more popular approaches. A basic outline of nested dissection is as follows:

(1) First, a graph is created based off of a sparse matrix in which rows and columns are represented by nodes and non-zero elements are represented by edges.

(2) Next, the graph is partitioned using a separator in order to divide the problem into smaller, easier to solve problems (divide and conquer)

(3) Recursively partition the graph to reduce the size of the subproblems

(4) Combine the subgraphs to obtain the reordered matrix

(5) Finally, perform decomposition on the newly ordered matrix

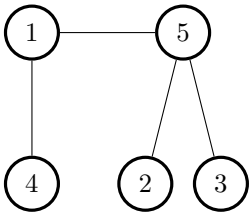
For a better understanding of Nested Dissection, we will separate this section into three subsections and discuss in detail each of its steps.

3.2.2 Creating a Graph from a Sparse Matrix

If the matrix is symmetric, the graph can be 1-dimensional. For example, the following matrix

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$

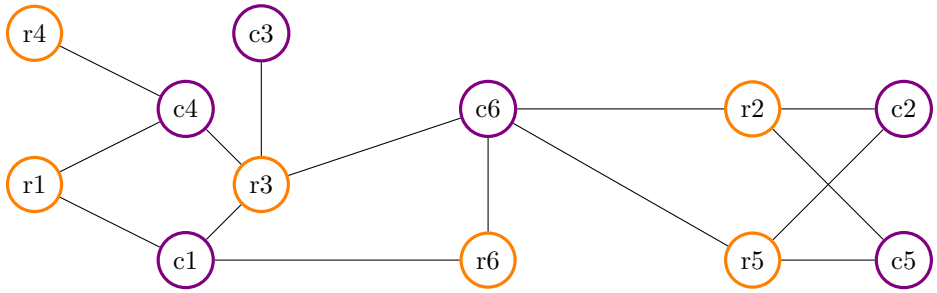
Results in the corresponding graph



There are several approaches for a nonsymmetric linear system. In this paper we will focus on the use of a hypergraph. To create a hypergraph, we let $G = (V, E)$ where V is a set of vertices and E is a set of edges. The rows and columns are each represented by a different vertex and the set of edges is made up of the nonzero entries of the matrix. For example, let A be the following matrix,
[7]

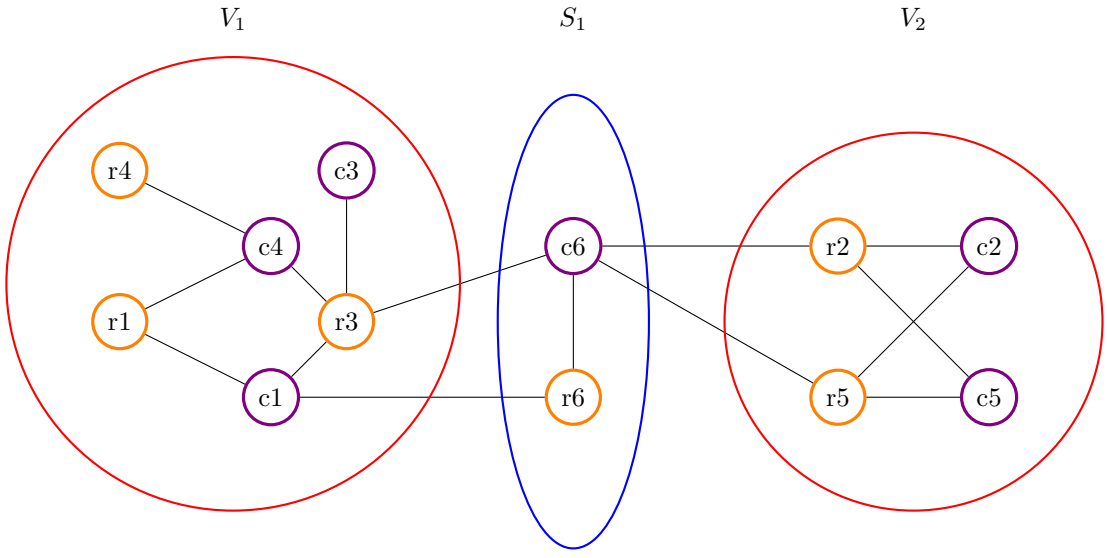
$$A = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Transforming this matrix results in the following hypergraph



3.2.3 Dividing the Graph Using Separators

Once we have created our graph, we can use an initial separator, S_1 to divide it into two unrelated parts. This involves separating the graph into two irreducible subgraphs. In general, the smaller the separator, the larger the reduction of fill-in. For the above graph, a separator, S_1 (given in blue) separates the two subgraphs, V_1 and V_2 (given in red).



To find an optimal separator is NP-hard which means nested dissection only finds a "good enough" separator, not necessarily an optimal one. Graph partitioning is a widely researched topic and as a result, there are many algorithms that can achieve a near optimal partition. One such approach proposed by Karypis and Kumar (1998) involves coarsening, partitioning, then uncoarsening the graph [1]. In most cases, the partitioning phase will be executed recursively on each of the subgraphs. This allows for partitioning in parallel where each of the k subgraphs are assigned to k different processors for even further improvement of computational efficiency. Note, however, that there has been shown to be a tradeoff between parallelism and fill-in [8]. In particular, by reducing parallelism, it is possible to reduce fill-in that elimination orders generate.

3.2.4 Form Reordered Matrix

Once the graph has been divided into distinct parts, using a separator, we can convert these subgraphs back into a reordered matrix. If the original graph has been divided into k parts, the reordered matrix will have k blocks of nonzeros along the diagonal and $k-1$ 'block borders.' Each of the k blocks represents the node relationships in a distinct subgraph and each of the $k-1$ 'block borders' represent the separators. For the initial matrix in section 3.2.2, the newly ordered matrix will take the form

$$A_{DBBD} = \begin{pmatrix} & 1 & 3 & 4 & 2 & 5 & 6 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 & 1 \\ 5 & 0 & 0 & 0 & 1 & 1 & 1 \\ 6 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

In this example, we have achieved a Doubly Bordered Block Diagonal (DBBD) Matrix that takes on the optimal arrowhead structure depicted in section 3.2.1. Here, the subgraph V_1 is represented by the 3x3 block in the upper left-hand side of the matrix. It shows the relationships between the nodes 1, 3, and 4, just like the V_1 subgraph. The other red square, depicts the subgraph V_2 and the relationships between the 2, and 5 nodes. Finally the blue border depicts the separator used to divide the original graph into V_1 and V_2. It is important to note that a DBBD matrix form is not necessary for a good reordering it is just one common format.

An important take away from this result is that the red sections of the matrix will be recursively decomposed whereas the blue sections will remain untouched. Once the recursive decomposition has finished, each block can be solved and put back together to form the solution. This is because each subgraph is a disjoint set of vertices and each vertex separator is a disjoint set of vertices, which means solving each set of vertices will not influence the result of another subgraph. Therefore, the sum of the solutions to these subgraphs is the solution to the entire graph. This observation also implies that most of the work in solving the system will be spent in processing the largest set of vertices. It turns out that the first vertex separator is the largest set of vertices. This fact will be important in section 5.

3.2.5 Factorization

Once the reordered matrix has been achieved, classic LU decomposition can be performed to obtain much better results than using the original matrix. To complete our example, the L and U matrices are given below

$$LU = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Here, L has 10 nonzero entries and U has 9 making them both clearly sparse.

4 Numerical Results

We will now take a look at how well the aforementioned sparse solvers perform relative to a regular LU decomposition. Our experiments for the Thomas algorithm will look at the difference in runtime between using an LU decomposition to solve the system versus using the Thomas algorithm. Since nested dissection produces a fill-reducing ordering, we will also compare the fill-in created from a regular LU decomposition with the fill-in created from nested dissection.

4.1 Results For the Thomas Algorithm

The first experiment that we will present is a comparison between the performance of the Thomas algorithm and a regular LU decomposition. For this experiment, we used the finite difference matrix obtained when discretizing the one-dimensional Laplace equation. We will give a brief (but not comprehensive) derivation of the finite difference matrix. Recall the Laplace equation:

$$\Delta u_{xx} + f(x) = 0 \implies \Delta u_{xx} = -f(x)$$

If on the interval of interest (a,b) we enforce the boundary conditions $u(a) = u(b) = 0$, then the Laplacian of u can be approximated by the following:

$$u_{xx}(x_i) \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}$$

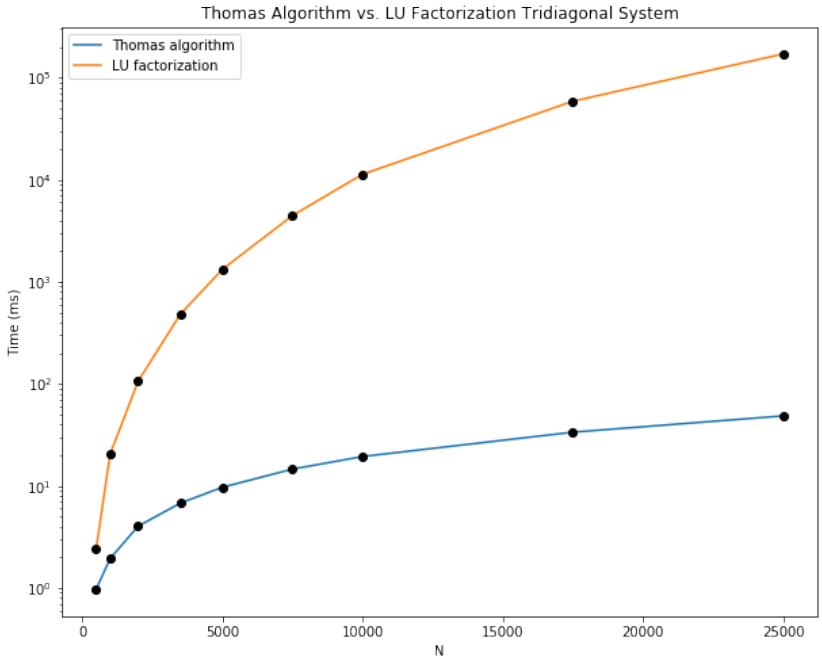
where

$$h = \frac{b-a}{n}$$

Equating $u_{xx}(x_i)$ to $-f(x_i)$ for all x_i, we have the following system of equations:

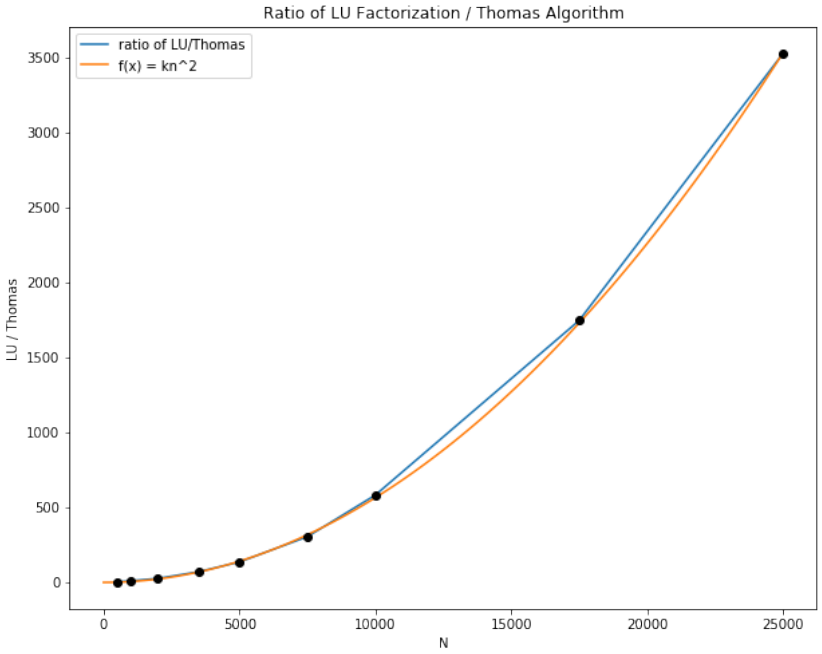
$$\frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & ... & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = - \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix}$$

Since this is a tridiagonal matrix, we can use the Thomas algorithm to solve the system. For our tests, we used nine tridiagonal matrices with $n = 500$, 1000, 2000, 3500, 5000, 7500, 10000, 17500, and 25000. Since we were not concerned with the solution, only the time to find the solution, our $\mathbf{f}$ vectors were randomly generated. The results are shown in the figure below:



Comparison of time complexity of the Thomas algorithm versus LU decomposition

We can also verify that the algorithms are behaving as expected by taking the ratio of runtime for LU and runtime the Thomas algorithm. Since performing an LU decomposition to solve a system is $\mathcal{O}(n^3)$ and the Thomas algorithm is $O(n)$, we should expect their ratio to be $c_1 n^3 / c_2 n = kn^2$, where c_1, c_2, and k are arbitrary constants. The following plot confirms this expectation:



Ratio of run time of LU factorization over run time of Thomas algorithm

As one can see, the ratio almost perfectly follows an n^2 curve. Having a $\mathcal{O}(n^2)$ speed up in computation is immense and makes the Thomas algorithm one of the best sparse solvers. Unfortunately, as we have mentioned before, this algorithm only applies to tridiagonal systems. In the case that our matrix is not tridiagonal, we

must rely on a more general approach in nested dissection. Our numerical results associated with this method are discussed in the next section.

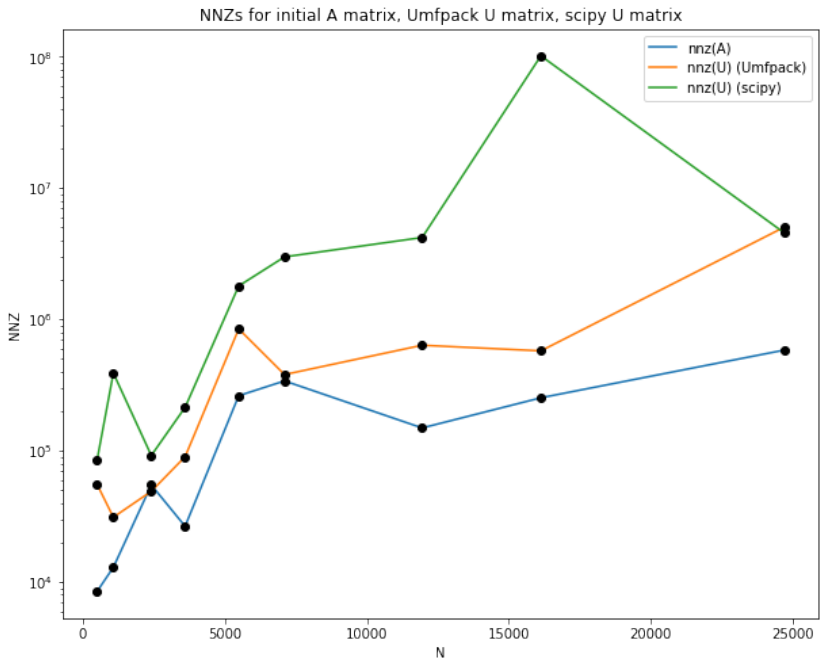
4.2 Results For Nested Dissection

In the following experiments, we used nine symmetric sparse matrices and nine unsymmetric sparse matrices. We randomly selected the sets of matrices so that there would be variance in the structure of the matrices. If we were to only choose one specific structure of matrices (such as banded matrices), our results would not be indicative of the performance of nested dissection as a whole. It should be noted that the primary nested dissection package that we used for our experiments was Umfpack (Unsymmetric-pattern MultiFrontal Package). So that our readers can better understand our results, we have supplied the images of all 18 matrices (9 symmetric, 9 unsymmetric) we used in our experiments in the Appendix under section 7.1. In referencing a particular matrix, we will use the convention SM 1 to refer to the first symmetric matrix, and USM 1 to refer to the first unsymmetric matrix. This convention extends to any number 1-9.

The set of results that we will present is an investigation into the reduction of number of nonzero entries from the U matrix created in a normal LU decomposition to the U matrix of the Umfpack LU decomposition. Our findings are summarized in the tables below and further illustrated by the corresponding plots. We supply the density of the initial matrix (A) for reference.

Comparison of NNZ for Symmetric Matrices				
Size (N)	% NNZ(A)	% NNZ SciPy LU	% NNZ Umfpack LU	Factor of Decrease in NNZ
500	3.39	33.9	22.2	1.53
1074	1.12	33.8	2.70	12.5
2410	.944	1.58	.845	1.87
3600	.205	1.65	.695	2.38
5489	.871	5.97	2.81	2.13
7102	.674	5.93	.754	7.87
11948	.104	2.95	.445	6.61
16129	.0973	39.1	.221	176
24696	.0957	.749	.826	.906

For the first seven matrices, we observe a decrease in the nnzs from the SciPy U matrix to the Umfpack U matrix by a factor in the range of 1.53 - 12.5. For SM 8, since its structure is approximately an arrowhead matrix pointing to the top left (which is the least optimal structure for avoiding fill-in), we expect to observe the greatest difference in nnzs between the SciPy U matrix and the Umfpack U matrix. This is certainly the case as the nnzs decreases by a factor 176. It is perhaps this example that best illustrates the power of a nested dissection reordering. Interestingly, for SM 9, we actually observe a slight increase in the nnzs. However, the matrix was already very sparse and so the increase in nnzs is a worthy trade for the reduction in time to compute the solution using nested dissection. Presented below is a plot visualizing the reduction of nnzs.

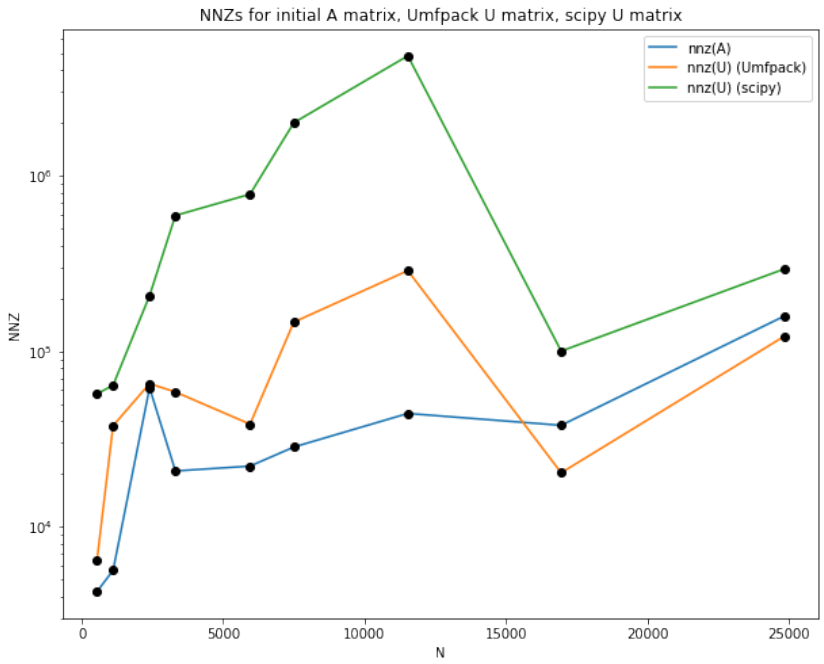


Comparison of NNZs for symmetric sparse matrices. The initial matrix denisties, the SciPy U matrix densities, and the Umfpack U matrix densities are shown on a log scale.

The plot allows us to see that Umfpack does pretty well in maintaining sparsity in the U matrix whereas the SciPy LU factorization can fail to produce a sparse decomposition. This is a good illustration of the importance of reordering a matrix before decomposing it. We will now take a look at the nnz reduction using Umfpack on unsymmetric matrices.

Comparison of NNZ for Unsymmetric Matrices				
Size (N)	% NNZ(A)	% NNZ(U) SciPy LU	% NNZ(U) Umfpack LU	Factor of Decrease in NNZ
541	1.46	19.6	2.22	8.81
1107	.461	5.22	3.07	1.70
2398	1.08	3.60	1.14	3.17
3312	.174	4.96	.490	10.1
5952	.0623	2.21	.108	20.4
7500	.0506	3.56	.260	13.7
11532	.0332	3.62	.216	16.8
16955	.0132	.0348	.00706	4.93
24842	.0257	.0476	.0197	2.42

Here the increase in sparsity ranges from 1.70 - 20.4 which is a pretty similar range compared to the symmetric matrices. The amount of fill-in that we observe in a normal LU factorization depends more on how closely the initial matrix follows an arrowhead structure up and to the left as opposed to the symmetry of the matrix. The accompanying plot illustrates the reduction of fill-in when using the Umfpack LU on unsymmetric matrices.



Comparison of NNZs for unsymmetric sparse matrices. The initial matrix denisties, the SciPy U matrix densities, and the Umfpack U matrix densities are shown on a log scale.

From the plot we can clearly see how nested dissection preserves the sparsity of the initial matrix (in certain cases it even improves upon the sparsity) and that the SciPy LU factorization is unable to avoid large amounts of fill-in.

5 Project Extension

Given that nested dissection is a technique aimed at reducing the fill-in that occurs while creating the LU factorization of a sparse matrix, it is natural to extend our research to runtime analysis. In essence, the purpose of finding a fill-reducing ordering via nested dissection is to minimize the time required to solve the system. In this section, we give an intuitive explanation of the time complexity of nested dissection and supply numerical results to illustrate the improvement in computational work.

5.1 Time Complexity of Nested Dissection

It is often the case that the graphs we construct from the initial sparse matrix in nested dissection is a planar (or near planar) graph. This is nice because planar graphs have an associated $n^{1/2}$-separator theorem associated with them. This theorem was stated and proven by Lipton and Tarjan [11] in 1979. The theorem is the following:

Theorem 5.1.1 ($n^{1/2}$ Separator) *Let G be any n-vertex planar graph having nonnegative vertex costs summing to no more than one. Then the vertices of G can be partitioned into three sets A, B, C such that no edge joins a vertex in A with a vertex in B, neither A nor B has total cost exceeding 2/3, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices.*

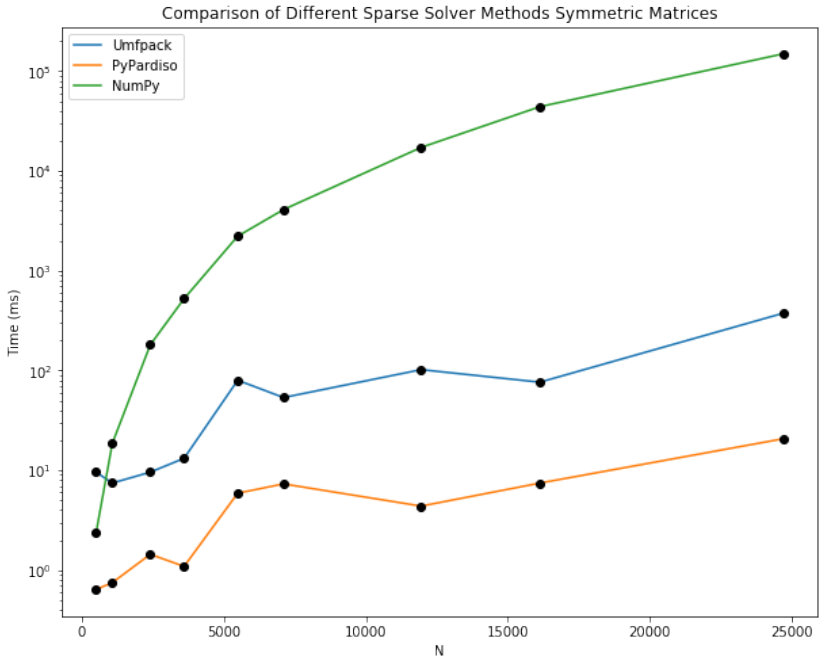
The main takeaway of this theorem for our purposes is that the vertex separator contains no more than $2\sqrt{2}\sqrt{n}$ vertices. This means that the largest vertex separator throughout the recursive decomposition of nested dissection will not exceed $2\sqrt{2}\sqrt{n}$ vertices. As discussed in section 3.2.4, processing the largest vertex separator is usually where most of the computational efforts are spent. In processing each set of vertices, we are actually inverting the corresponding matrix to find the solution to the subproblem it defines. As covered in this paper, solving a system via Gaussian elimination is a $\mathcal{O}(n^3)$ operation. But since n in this case no

greater than $2\sqrt{2}\sqrt{n}$ (as given by Lipton and Tarjan), solving this system has cost $\mathcal{O}(n^{3/2})$. Because this is the largest system after performing nested dissection, the overall complexity of nested dissection is $\mathcal{O}(n^{3/2})$ (in the case that the initial problem defines a planar graph). In the case that the initial matrix defines a non-planar graph, there has been little work done in determining the efficiency of nested dissection. However, since sparse systems usually result in planar graphs, the time complexity of $\mathcal{O}(n^{3/2})$ is often a good representation of the performance of nested dissection.

5.2 Empirical Results for Runtime Analysis

We now present a set of experiments aimed at comparing the runtime of solving the 18 linear systems. We tested two sparse solver packages, Umfpack and PyPardiso, and compared the performance of these sparse solvers to the built in linear algebra solver in NumPy. This built in command calls the LAPACK routine _gesv which performs an LU decomposition to solve the system. We conducted our tests on symmetric matrices and unsymmetric matrices and summarized our results in a table for the two types of matrices and visualized our findings in a corresponding plot. The results are shown below.

Time to Compute Solution Symmetric Matrices (ms)				
Matrix	Size (N)	NumPy	Umfpack	PyPardiso
SM 1	500	2.36	9.57	0.640
SM 2	1074	18.7	7.46	0.750
SM 3	2210	181	9.58	1.44
SM 4	3600	525	13.2	1.09
SM 5	5489	2220	79.9	5.90
SM 6	7102	4100	53.6	7.31
SM 7	11948	17200	102	4.38
SM 8	16129	44000	76.4	7.45
SM 9	24696	149000	375	20.7

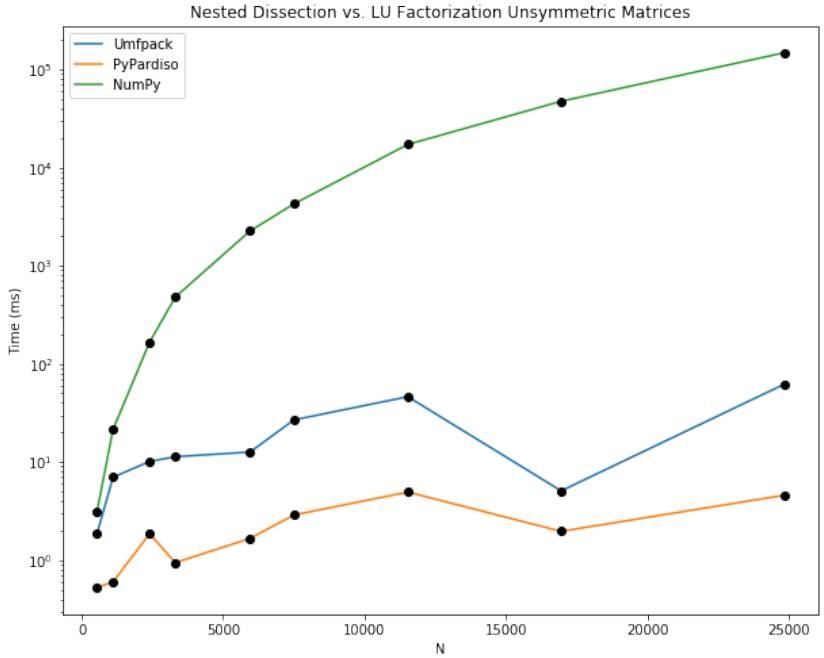


Comparison of runtimes for two sparse solver packages, Umfpack and PyPardiso, and the standard NumPy routine applied to symmetric matrices

From the plot, we can clearly see just how much faster sparse solvers are compared to an LU factorization. When N is close to 25000, the time to compute the solution using Umfpack takes 375 ms and only 20.7 ms

using PyPardiso. When solving via an LU factorization, obtaining the solution takes two and a half minutes. One can imagine that in the "real world", when we have a system with over a million rows and columns, attempting to solve a system with an LU factorization would take many days, possibly even months. But when a sparse solver is applied to the system, the solution can be obtained in only a matter of minutes.

Time to Compute Solution Unsymmetric Matrices (ms)				
Matrix	Size (N)	NumPy	Umfpack	PyPardiso
USM 1	541	3.15	1.87	0.524
USM 2	1107	21.5	6.98	0.600
USM 3	2398	166	10.1	1.85
USM 4	3312	480	11.3	.941
USM 5	5952	2260	12.6	1.66
USM 6	7500	4260	26.8	2.88
USM 7	11532	17200	46.2	4.93
USM 8	16955	47400	5.07	1.96
USM 9	24842	148000	61.9	4.58



Comparison of runtimes for two sparse solver packages, Umfpack and PyPardiso, and the standard NumPy routine applied to unsymmetric matrices

From our tests we can see that the time to solve an unsymmetric system versus a symmetric system is roughly equivalent, especially in the case of the NumPy routine. The performance on unsymmetric matrices when using the sparse solver packages is slightly better than symmetric ones which we suspect is a result of the unsymmetric matrices having more sparse LU decompositions (refer to the comparison of NNZ tables). There is nothing in the nested dissection algorithm that favors unsymmetric matrices over symmetric ones which leads us to believe it is more of a coincidence than anything that unsymmetric matrices performed better. Overall, the time to compute a solution using the sparse solver packages on the two types of matrices is still on the same order of magnitude so the relative difference in computation time is not overly significant.

6 Conclusion and Further Exploration

In response to our (implicity defined) initial question, we found that yes, we can take advantage of the nonzero elements in sparse systems. In particular, through analysis of solve time and fill in, we found that solving

sparse matrices using nested dissection was far more efficient than the normal linear algebra techniques. It was expected that these direct solving methods would perform faster, but the degree to which they sped up computation time was still surprising. The results of this paper can be extended to solving actual sparse systems that arise in the real world. Furthermore, our results make it clear that solving large sparse systems would be impractical without the use of sparse solvers. In future exploration, we would like to continue our discussion of nested dissection by analyzing different methods for improving upon the algorithm. One such topic for improvement is partitioning techniques, as finding an optimal partition is one of the more computationally intensive aspects of nested dissection. Additionally, we would like to look into the importance of elimination trees and the role of cholesky factorization in solving sparse symmetric positive definite matrices.

References

[1] Karypis, George, and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs." SIAM J. Scientific Computing, 1998, https://www.cs.utexas.edu/ pingali/CS395T/2009fa/ papers/metis.pdf. Accessed 13 Dec. 2021.

[2] Bindel. Matrix Computations (CS 6210). 2016, https://www.cs.cornell.edu/ bindel/class/cs6210-f16/lec/2016-09-23.pdf.

[3] Brainman, Igor, and Sivan Toledo. "Nested Dissection Orderings for Sparse LU with Partial Pivoting." SIAM J. Matrix ANAL. APPL., http://www.cs.tau.ac.il/ stoledo/Pubs/wide-simax.pdf. Accessed 13 Dec. 2021.

[4] O'Connor, Derek. An Introduction to Sparse Matrices. 1984, http://www.irishmathsoc.org/nl15/nl15_6-30.pdf.

[5] Boman, Erik G, and Michael M Wolf. "A Nested Dissection Approach to Sparse Matrix Partitioning for Parallel Computations." 2007, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.216.6351&rep=rep1&type=pdf. Accessed 13 Dec. 2021.

[6] Gilbert, John R, and Robert Endre Tarjan. The Analysis of Nested Dissection Algorithm. Springer-Verlag, 1987, https://link.springer.com/content/pdf/10.1007/BF01396660.pdf, Accessed 13 Dec. 2021.

[7] Ben Khaled - El Feki, Abir & Ben Gaid, Mongi & Simon, Daniel. (2013). Parallelization Approaches for the Time-efficient Simulation of Hybrid Dynamical Systems: Application to Combustion Modeling. 10.13140/2.1.4034.7847.

[8] Boman, Erik G., and wolf, michael m. A Nested Dissection Partitioning Method for Parallel Sparse Matrix-Vector Multiplication.. United States: N. p., 2013. Web. doi:10.1109/HPEC.2013.6670333.

[9] Engwer, C., Falgout, R. D., and Yang, U. M. Stencil computations for PDE-based applications with examples from DUNE and hypre. United States: N. p., 2017. Web. doi:10.1002/cpe.4097.

[10] Khaira, Manpreet S, et al. "Nested Dissection: A Survey and Comparison of Various Nested Dissection Algorithms." Carnegie Mellon University, 1992, https://www.cs.cmu.edu/ glmiller/Publications/CMU-CS-92-106R.pdf. Accessed 13 Dec. 2021.

[11] Lipton, Richard J, and Robert Endre Tarjan. "A Separator Theorem for Planar Graphs." SIAM J. APPL MATH, Apr. 1979.

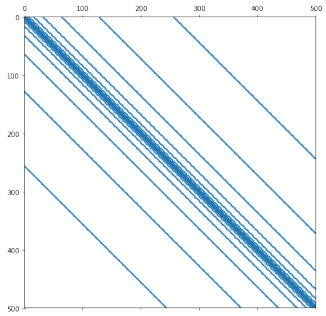
[12] W. T. Lee. "Tridiagonal Matrices: Thomas Algorithm." University of Limerick, Web. http://www.industrial-maths.com/ms6021_thomas.pdf. Accessed 13 Dec. 2021

[13] Yao Zhang, Jonathan Cohen, and John D. Owens. 2010. Fast tridiagonal solvers on the GPU. SIGPLAN Not. 45, 5 (May 2010), 127–136. Doi:https://doi.org/10.1145/1837853.1693472

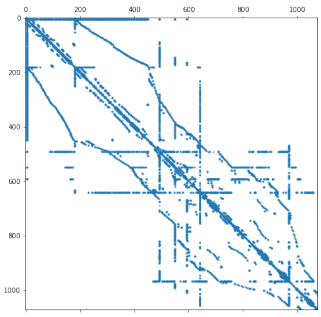
7 Appendix

7.1 Sparse Matrices

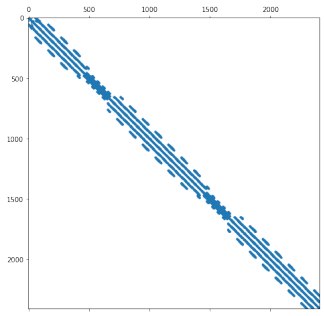
7.1.1 Symmetric Matrices



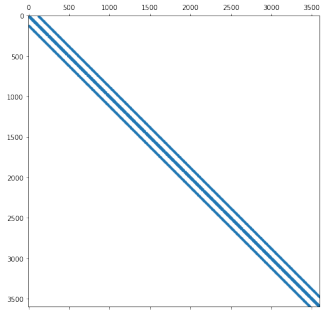
SM 1 (n = 500)



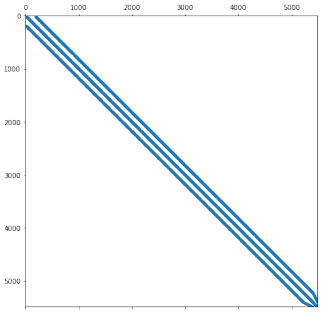
SM 2 (n = 1074)



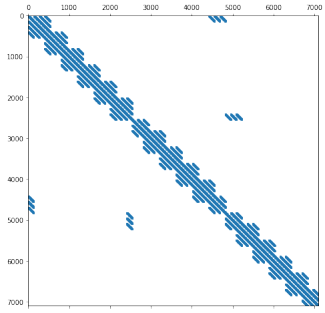
SM 3 (n = 2410)



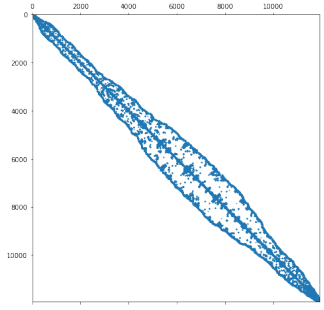
SM 4 (n = 3600)



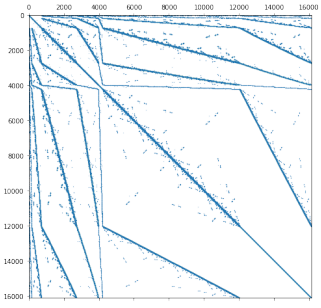
SM 5 (n = 5489)



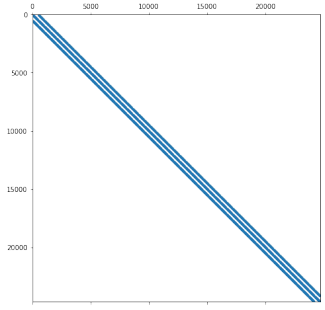
SM 6 (n = 7102)



SM 7 (n = 11948)

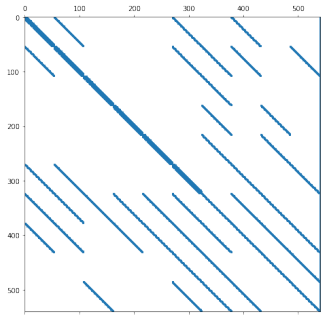


SM 8 (n = 16129)

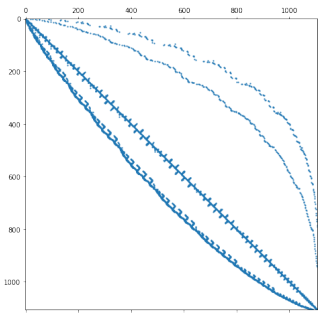


SM 9 (n = 24696)

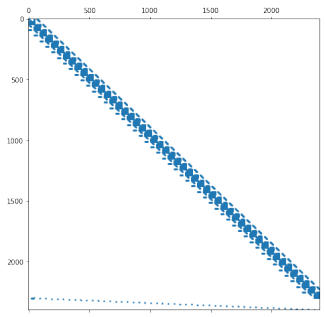
7.1.2 Unsymmetric Matrices



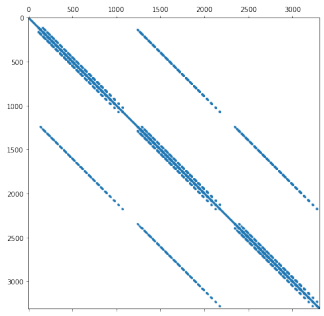
USM 1 (n = 541)



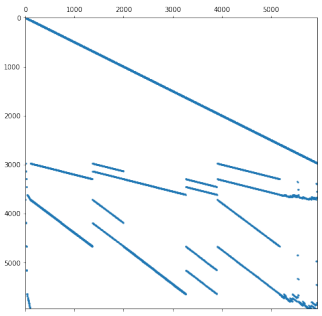
USM 2 (n = 1107)



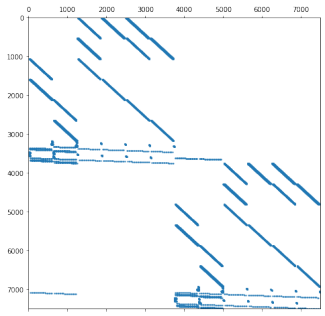
USM 3 (n = 2398)



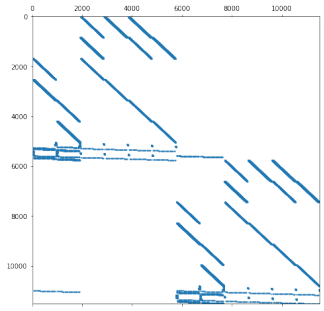
USM 4 (n = 3312)



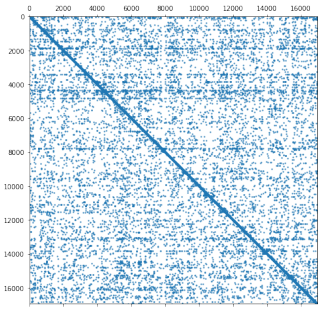
USM 5 (n = 5932)



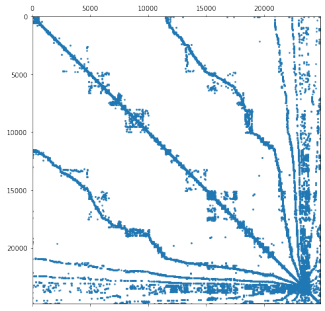
USM 6 (n = 7500)



USM 7 (n = 11532)



USM 8 (n = 16955)



USM 9 (n = 24842)

7.2 Thomas Algorithm Subroutine (Python)

```
def Thomas(a, b, c, d):
    nf = len(d) # number of equations
    ac, bc, cc, dc = map(np.array, (a, b, c, d)) # copy arrays
    for i in range(1, nf):
        mc = ac[i-1]/bc[i-1]
        bc[i] = bc[i] - mc*cc[i-1]
        dc[i] = dc[i] - mc*dc[i-1]

    x = bc
    x[-1] = dc[-1]/bc[-1]

    #backsolve
    for j in range(nf-2, -1, -1):
        x[j] = (dc[j]-cc[j]*x[j+1])/bc[j]

    return x
```