TRIPOLL IN GRAPHBLAS

An Undergraduate Research Scholars Thesis

by

ABEER WAHEED

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Tim Davis

May 2022

Major: Computer Engineering – Electrical Track

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Abeer Waheed, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

Page

CHAPTERS

ABSTRACT

TriPoll In GraphBLAS

Abeer Waheed
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Tim Davis
Department of Computer Science and Engineering
Texas A&M University

Creating Graph algorithms as fast as possible, in as little time as possible, is a key goal for countless data scientists. To this end, GraphBLAS can be used. GraphBLAS is a standard that defines sparse matrix operations on semirings. When these operations are applied to matrices, they work in a similar manner to traditional Graph computations, and can further be used to write Graph Algorithms and solve Graph problems. This allows Graph Algorithms to leverage existing parallel techniques found in the heavily researched and older field of linear algebra, allowing for faster development times, lower complexity and strong performance. Using SuiteSparse: GraphBLAS (an implementation of the GraphBLAS standard), a GraphBLAS implementation of the TriPoll algorithm is built, and the process and results are examined. The goal is not necessarily to build the fastest algorithm, but rather to see what GraphBLAS can achieve while balancing complexity and speed. TriPoll is an algorithm that is capable of surveying triangles (3-cycles) in Graphs with metadata on them. TriPoll was chosen to be implemented in GraphBLAS because it was recently published, has useful functionality, and is relevant, especially for social networks.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Tim Davis, for his guidance and support throughout the course of this research.

Thanks also go to my friends, colleagues, and the department faculty and staff for making my time at Texas A&M University a great experience.

The research papers used for TriPoll in GraphBLAS were provided by Dr. Tim Davis, and some Myself.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Funding was provided by NVIDIA, Intel and Redis for this research.

1. INTRODUCTION

1.1 GraphBLAS Introduction and Overview

GraphBLAS is a standard that defines matrix and vector operations built using semirings, which can be applied to matrices and vectors to mimic traditional Graph operations. Kepner and Gilbert [1] provide a deeper understanding of how matrix computaions can map to Graph Algortihms. Using GraphBLAS allows users to leverage parellel techniques found in linear algebra to improve preformance while keeping development time down. Using the GraphBLAS standard has several advantages, including, but not limited to, opaque data types (which allow the user to focus on the algorithm rather than the minute details, allowing users to apply bulk operations on Graphs without worrying about each individual node and edge), and portibility (meaning that as a standard, if a new implementation of GraphBLAS is released that happens to be faster than the current implementation, the code should not have to be changed). These advantages allow users of GraphBLAS to provide a high level view of an algorithm in linear algebra, which allows for a reduction in development time and still provides competetive performance. Dr. Tim Davis showed the versatility of GraphBLAS in [4], where Dr. Davis was able to solve the Sparse Deep Neural Network Graph Challenge in 20 minutes of programmer time. GraphBLAS can, and already is, being used to solve large scale computational problems.

1.1.1 Adjacency Matrices

One of the most crucial elements of understanding GraphBLAS, and using linear algebra on Graphs, is understanding adjacency matrices. Traditionally, Graphs are represented using adjacency lists, which provide a list of all the nodes that are connected to a single node through edges. Adjacency matrices are another way to represent Graphs in the form of matrices, and can be done for all Graphs regardless of type. Given an adjacency matrix A, if A (i, j) has a value, then there exists an edge from i to j in the graph. In this context, i represents the index of the row, j represents the index of the column, and A (i, j) is the value at row i and column j. So, for example, in a Graph, if there is an edge from Node 3 to Node 4, then in the adjacency matrix there should be a corresponding value at row 3, column 4 (A (3,4)). This can be easily extended to edge weights, where the value of the matrix entry A (i, j) represents the weight of the edge from i to j. If A (i, j) is 0, then the edge still exists, but its value is 0. Only when A (i, j) does not exist (has no value) can it be said that no edge from i to j exists. A value in A (i, j) does not imply that A (j, i) is present, which allows matrices to represent graphs that have direction. Matrices also need not be squares (the number of rows need not equal the number of columns), which can be useful when analyzing certain special graphs, such as bipartite graphs (among other Graphs). When analyzing larger Graphs, a question that may arise is how GraphBLAS stores such large Graphs. Graphs can have millions, or even billions of edges, so how they are stored should be of concern, as it may seem that matrices require extensive space to account for entries that do not exist (edges between nodes that do not exist). The answer lies in GraphBLAS's use of opaque data structures. Edges that are not present in the Graph are not represented in the underlying data structure used to create the matrix, which allows for encoding of larger Graphs into GraphBLAS. One of the key advantages of GraphBLAS is that the underlying data structure does not matter to the user, and has instead been handled by the creator of the GraphBLAS implementations.

1.2 GraphBLAS Object Types

GraphBLAS includes 9 types of objects, which can be used in mathematical operations. A full explanation of the types can be found in the GraphBLAS Specification [2]. The 9 types of objects are as follows:

1. Types (GrB_Type)
2. Unary Operators (GrB_UnaryOp)
3. Binary Operators (GrB_BinaryOp)
4. Select Operators (GrB_Select)
5. Monoids (GrB_Monoid)
6. Semirings (GrB_Semiring)
7. Descriptors (GrB_Descriptor)
8. Vectors (GrB_Vector)
9. Matrices (GrB_Matrix)

1.2.1 Types (GrB_Type)

The predefined, built-in types for GraphBLAS are as follows: GrB_BOOL, GrB_INTx, GrB_UINTx, GrB_FP32, and GrB_FP64, where x represents 16, 32, or 64 bits. Any of these types, along with most user defined types (as long as they are static and contiguous) can be used when creating Matrix or Vector objects.

1.2.2 Unary Operators (GrB_UnaryOp)

A Unary Operator is an operator which acts on elements of GraphBLAS objects (Matrices or Vectors). It acts as a function with a single input, z = f(x) (z = -x, or z = x*x for example). GraphBLAS has several built-in Unary Operators, and the SuiteSparse implementation extends these Operators with even more that do not fit in the spec. Users can also create their own Unary Operators.

1.2.3 Binary Operators (GrB_BinaryOp)

Similar to Unary Operators, a Binary Operators also acts on elements of GraphBLAS objects. The difference is that Binary Operators require two inputs, and act as a function z = f (x, y) (z = x+y, or z = x-y for example). GraphBLAS has several built-in Binary Operators, and the SuiteSparse implementation extends these Operators with even more built-in Operators. Users can also create their own Binary Operators.

1.2.4 Select Operators (GrB_Select)

A select Operator chooses whether or not to keep a certain value in a GraphBLAS object. For example, selecting the lower triangular part of a Matrix, a useful and essential operation in many algorithms, can be done using a GrB_Select operation.

1.2.5 Monoids (GrB_Monoid)

Monoids are Binary Operators (z = f (x, y)) with additional constraints, those being they must be associative, commutative, the types of z, x, y must be the same, and the identity value must follow the pattern f (x, c) = f (c, x) = x, where c is the identity value for the specific monoid. For example, multiplication is a monoid, where the identity value is 1 since x*1 = 1*x = x. User-created monoids are also possible when given an operator and an identity value that follow the above constraints.

1.2.6 Semirings (GrB_Semiring)

Semirings may well be the most important part of the GraphBLAS types when trying to create a Graph algorithm using GraphBLAS. It consists of an additive monoid and a multiplicative operator that is used in the place of a matrix multiply. Traditionally, a matrix multiply is the dot product of the rows and columns of two matrices, where the rows of the first matrix are multiplied by the corresponding elements in the column of the second matrix, and then summed together to get a single element in the resulting matrix. However, the Semiring alters the traditional matrix multiply by replacing the multiplication with a new multiplicative operator, and the addition with an additive monoid. The traditional matrix multiplication becomes the plus-times semiring when thinking in terms of semirings. The multiplicative operator that replaces the scalar multiplication found in the plus-times semiring can be any binary-operator, even user defined ones, so long as the output type matches the type of the additive monoid. Not even counting user defined semirings, a large number of unique semirings can be created and used on Matrices.

The use of these semirings usually corresponds to useful Graph operations. For example, using the min-plus semiring, which replaces the traditional addition operation with the minimum operation, and the traditional multiplication operation with the plus operation, results in corresponding entries in the matrix being added, and the minimum being placed in the resulting matrix. This turns out to be quite useful for shortest path algorithms. Another example would be the Matrix-Vector multiply, in which the neighbors of a node (the nodes that a single node can reach through an edge) can be found and, if built upon, can be used to create the Breadth-First-Search algorithm. These, and many more Graph algorithms, can be built using user-defined or built-in semirings.

1.2.7 Descriptors (GrB_Descriptor)

The Descriptor type can be passed to a GraphBLAS operation to change the input or the output of an operation, such as transposing the input matrix.

1.2.8 Vectors (GrB_Vector)

A vector is the traditional definition of a vector, which essentially corresponds to an array in C. Many of the previously defined operations described can be applied to Vectors.

1.2.9 Matrices (GrB_Matrix)

A matrix is the traditional definition of a matrix, which is essentially a set of rows and columns with values in those rows and columns. Many of the previously defined operations described can be applied to Matrices.

1.3 TriPoll Introduction/Overview

The TriPoll algorithm is an algorithm that surveys 3-cycles (triangles) in Graphs with embedded metadata and returns the metadata on those triangles. This metadata can take the form of anything from a simple number to a string of characters. The reference implementation [5] can be seen for more specific information. Triangle counting, a special case of the TriPoll algorithm where there is no metadata on the edges (or the edge metadata is 1), is already difficult to enumerate, and often times approximations are made instead. When working with metadata on edges though, finding every triangle and providing the metadata may be necessary.

1.3.1 TriPoll Implementation in GraphBLAS

The TriPoll Algorithm implemented in GraphBLAS takes in a Graph with Metadata on its edges, and returns the data of each triangle to the user, who can run a variety of different functions on it. The algorithm first reads in the problem, making sure that the input is valid, then finds the DODGr of the input Graph. The DODGr is an ordering of the vertices by Degree, which helps with computation time. Using the DODGr, the number of triangles is found through a Matrix Multiplication and a reduction operator. Then, the output matrix is used to find the nodes of the triangle. Currently, there are three implementations that have been created in

GraphBLAS to find the nodes of the triangles. Regardless of the method used, the metadata of the found triangle is extracted and given to the user, who can run any function they would like on them.

2. METHOD/STEPS

2.1 Steps

Here, the steps of the algorithm will be laid out. First, the input is checked to make sure it complies with the rules. Then, the DODGr is found in order to reduce computation time. Then, the number of triangles is found, and finally, the metadata of those triangle are found.

The steps are as follows:

1. Input Check
2. DODGr find
3. Triangle Count
4. Triangle Metadata find

2.1.1 Input Check

2.1.1.1 Introduction

The purpose of Invalid Input Validation is to make sure that the code is robust and responds to improper input properly, so the user can be informed of the issue quickly and easily. It also makes sure that the algorithm is able to properly free all of its data. This is especially important in GraphBLAS, as it allocates much of its initial data on startup. It also prevents the case where the algorithm is able to run, yet the invalid inputs produce the wrong results. Since the purpose of this code is to eventually be included in the LAGraph library, it is important that the code be robust and provide the correct results.

2.1.1.2 Details

Input validation is able to be done relatively simply because LAGraph already has a built-in read function. This function takes in a Matrix Market File, and does all the relevant error checks on it. LAGraph also has a flag that can detect if a Graph is directed, and essentially make it undirected by adding the missing edges. If the Matrix Market File does not exist, the function outputs a file not found error. If the file is not in the proper Matrix Market File format, the function will give an error and status number that can be used to find additional information.

2.1.2 DODGr Find

2.1.2.1 Introduction

The DODGr of a graph is designed to take an undirected graph and make it directed by comparing the degrees (which is just the number of edges connected to that node) of the two nodes on each edge, and removing edges from nodes connecting high degrees to low degrees. This is done mainly for two reasons: reducing computation by removing additional ways to count triangles, and forcing triangles into a single, easy to find style. If a graph is left undirected, there are technically six ways that it can be counted, resulting in, at the very least, more computation than necessary, and at the worst an inaccurate triangle count. The DODGr forces a triangle (with nodes P, Q, R) to have edges (P, Q), (P, R) and (Q, R). Figure 2.1 below shows the form of all triangles in the DODGr Graph. This allows for only one way to count each triangle.

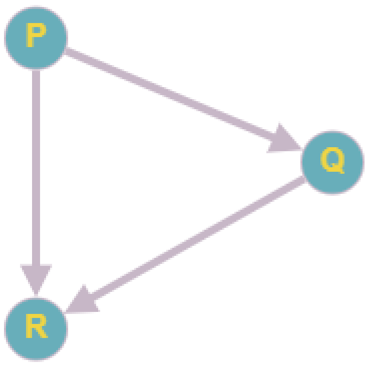
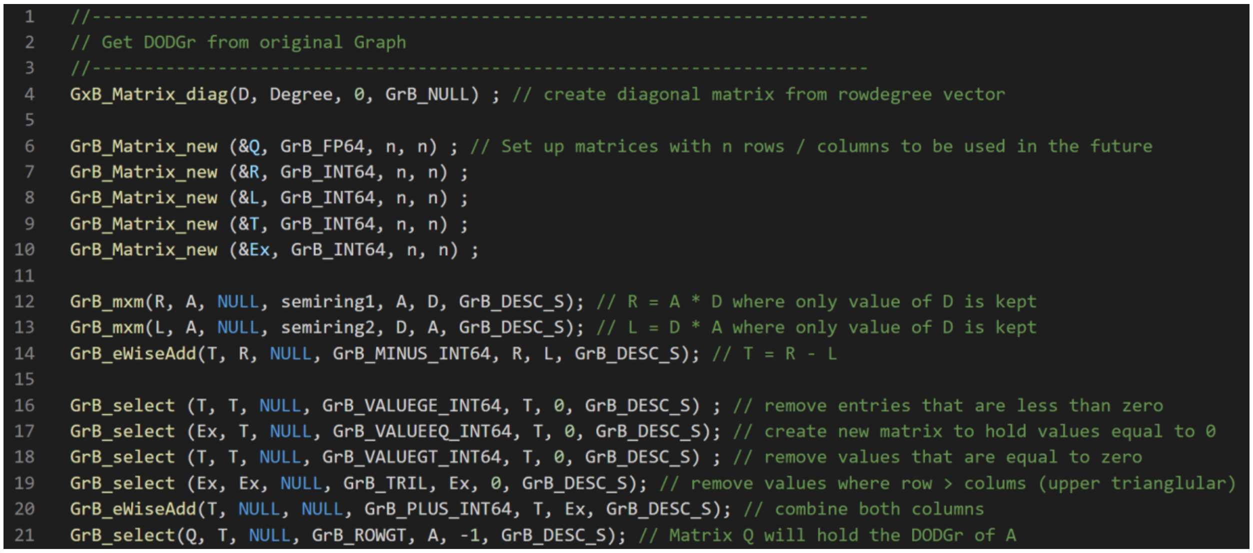


Figure 2.1: DODGr Triangle Example

2.1.2.2 Details

At a high level, the DODGr compares the degrees of the nodes on every edge, and removes the edge from the higher degree node to the lower degree node. In the case where the nodes have the same degree, the DODGr needs a determinate way to choose between the nodes. In the reference implementation of TriPoll, this was done by using a determinate hash function. In the GraphBLAS implementation, this was done by choosing the upper Triangular portion of the input Graph to break ties. Since Graphs with self-edges (edges that go to themselves) are not allowed (and are useless when working with triangles), this works as an ordering for all nodes. This selection of nodes is done by taking the row degree of each row (where the row degree is the number of entries in each row) in the initial input matrix, and then placing it in a vector. That vector is then placed on the diagonal of an empty matrix, creating a new matrix with values only on its diagonal. Then this new matrix is multiplied by the original input matrix of the Graph using the ANY semiring, which essentially ignores the value of the original input and only uses the value of the new matrix. Doing this once with the original matrix multiplied by the new matrix, and again with the new matrix multiplied by the original matrix, creates 2 matrices, one where each entry A (i, j) holds the degree of its row (i), and the other where each entry A (i, j) holds the degree of its column (j). Then, simply subtracting the entries in the matrices and dropping the negatives removes the edges from the high degrees to the low degrees. In order to handle the case where the degrees of the 2 nodes are equal (or, in this case, where the value after the subtraction is 0), the DODGr essentially keeps the edges where the node degrees are equal if they are in the lower triangular part of the matrices (which are just the edges where the row number is greater than the column). It then uses this new matrix as a mask (a mask only allows values that are in the mask to be seen in the matrix being masked, and drops all values that are not in the mask) for the input matrix of the Graph. The code for finding the DODGr is shown in Figure 2.2 below:



```
//--------------------------------------------------------------------------
// Get DODGr from original Graph
//--------------------------------------------------------------------------
GxB_Matrix_diag(D, Degree, 0, GrB_NULL) ; // create diagonal matrix from rowdegree vector

GrB_Matrix_new (&Q, GrB_FP64, n, n) ; // Set up matrices with n rows / columns to be used in the future
GrB_Matrix_new (&R, GrB_INT64, n, n) ;
GrB_Matrix_new (&L, GrB_INT64, n, n) ;
GrB_Matrix_new (&T, GrB_INT64, n, n) ;
GrB_Matrix_new (&Ex, GrB_INT64, n, n) ;

GrB_mxm(R, A, NULL, semiring1, A, D, GrB_DESC_S); // R = A * D where only value of D is kept
GrB_mxm(L, A, NULL, semiring2, D, A, GrB_DESC_S); // L = D * A where only value of D is kept
GrB_eWiseAdd(T, R, NULL, GrB_MINUS_INT64, R, L, GrB_DESC_S); // T = R - L

GrB_select (T, T, NULL, GrB_VALUEGE_INT64, T, 0, GrB_DESC_S) ; // remove entries that are less than zero
GrB_select (Ex, T, NULL, GrB_VALUEEQ_INT64, T, 0, GrB_DESC_S); // create new matrix to hold values equal to 0
GrB_select (T, T, NULL, GrB_VALUEGT_INT64, T, 0, GrB_DESC_S) ; // remove values that are equal to zero
GrB_select (Ex, Ex, NULL, GrB_TRIL, Ex, 0, GrB_DESC_S); // remove values where row > colums (upper trianglular)
GrB_eWiseAdd(T, NULL, NULL, GrB_PLUS_INT64, T, Ex, GrB_DESC_S); // combine both columns
GrB_select(Q, T, NULL, GrB_ROWGT, A, -1, GrB_DESC_S); // Matrix Q will hold the DODGr of A
```

Figure 2.2: DODGr Code

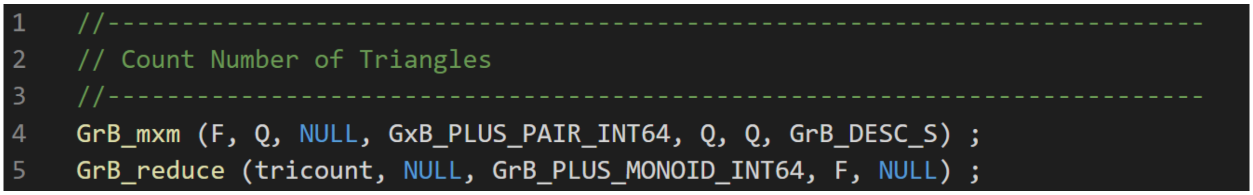
2.1.3 Triangle Counting

2.1.3.1 Introduction

Triangle counting is the lesser of the two main purposes of this algorithm. Triangle counting has a wide range of uses, especially in data science, where it can be used to find small, cohesive communities. Many algorithms that have been previously created focus solely on just this part, but this algorithm accomplishes a larger goal. Nevertheless, the code for triangle counting in particular demonstrates the fast yet less complex goal that GraphBLAS excels at.

2.1.3.2 Details

Due to the DODGr found in the previous Section, the algorithm for triangle counting is extremely easy and efficient. The code for triangle counting simply multiplies the DODGr by its transpose (the row and the columns are flipped) using the ANY_PAIR semiring (which is a normal matrix multiply where the values are considered to be 1). This gives a single edge of every triangle, specifically the edge from P to R. Then, it reduces the output matrix into a single number by adding up all the numbers in the output matrix, and that number ends up being the triangle count. This code is efficient and extremely simple, as can be seen in Figure 2.3 below:



```
//--------------------------------------------------------------------------
// Count Number of Triangles
//--------------------------------------------------------------------------
GrB_mxm (F, Q, NULL, GxB_PLUS_PAIR_INT64, Q, Q, GrB_DESC_S) ;
GrB_reduce (tricount, NULL, GrB_PLUS_MONOID_INT64, F, NULL) ;
```

Figure 2.3: Triangle Count Code

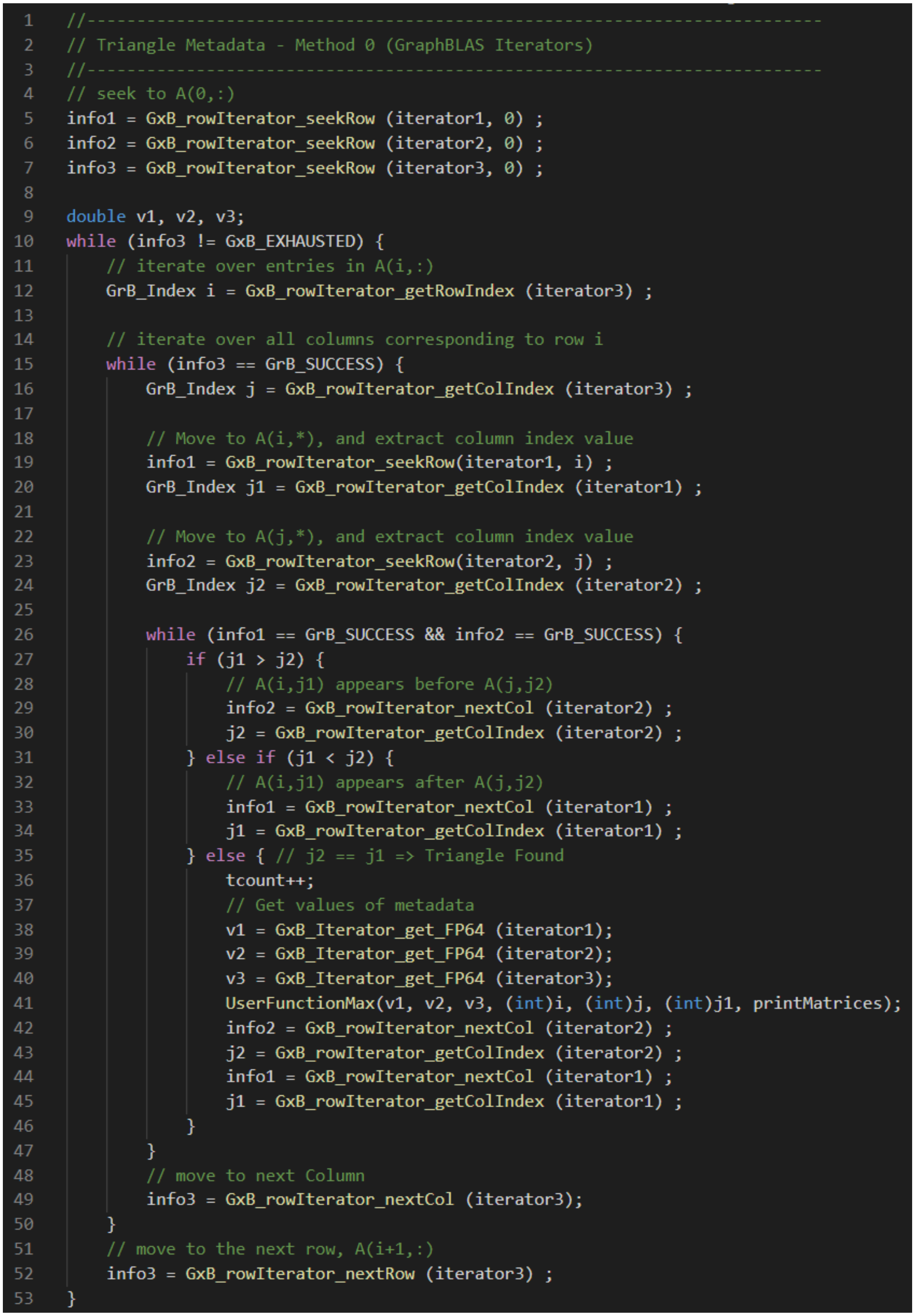
2.1.4 Triangle Metadata

2.1.4.1 Introduction

Finding the Triangle Metadata is the main purpose of this algorithm. Triangle metadata provides much more information to the user than only the triangle count, and has an even wider range of applications in data science. This, of course, was the hardest part of this algorithm, but is also the most useful and differentiating part of this algorithm. There are three implementations used to find triangle data, two that use purely GraphBLAS (Methods 0 and 1), and another that exports the matrix to arrays (Method 2).

2.1.4.2 Details for Method 0

Method 0 uses iterators (which are currently not in the spec, but are included in SuiteSparse) to iterate through the matrix to find the nodes that make up a triangle. First, three iterators are created and initialized to the first row of the matrix. One of these iterators (iterator3 for the code in Figure 2.4) is used to iterate across every row and column of the matrix. For every entry found by iterator3, the column number (j) and the row number (i) are extracted. Then, another of the three iterators (iterator1 in Figure 2.4) is set to iterate starting at row i. The final iterator (iterator2 in Figure 2.4) is set to the start iterating at the row corresponding to the column of iterator3 (row j). Then, the two iterators corresponding to row i and row j (iterator1 and iterator2 in Figure 2.4) are set to iterate across each of their respective rows. As they are iterating across their rows, the column values are being extracted and compared. If one of the iterator’s column values is lower than the other, then only that iterator is moved to the next column, and this continues until one of the two iterators reaches the end of the its row. A triangle is found when the two iterators are at the same column value. This is because, using iterator1, there is a known edge (i, j). Using iterator2, there is another edge (j, j2). Using iterator3, there is a final edge (i, j1). If j1 = j2, then a triangle has been found, where P = i, Q = j, and R = j1 = j2. After a triangle is found, the metadata at each of the edges is extracted, and the metadata along with the nodes for the triangle are given to the user defined function. After either one of the iterators reaches the end of the row, the iterator used to iterate across the entire matrix (iterator3 in Figure 2.4) is moved to the next column value, and the process is repeated. When the iterator reaches the end of the row, it is moved to the next row until the entire matrix has been iterated across. The code for Method 0 is shown in Figure 2.4 below:

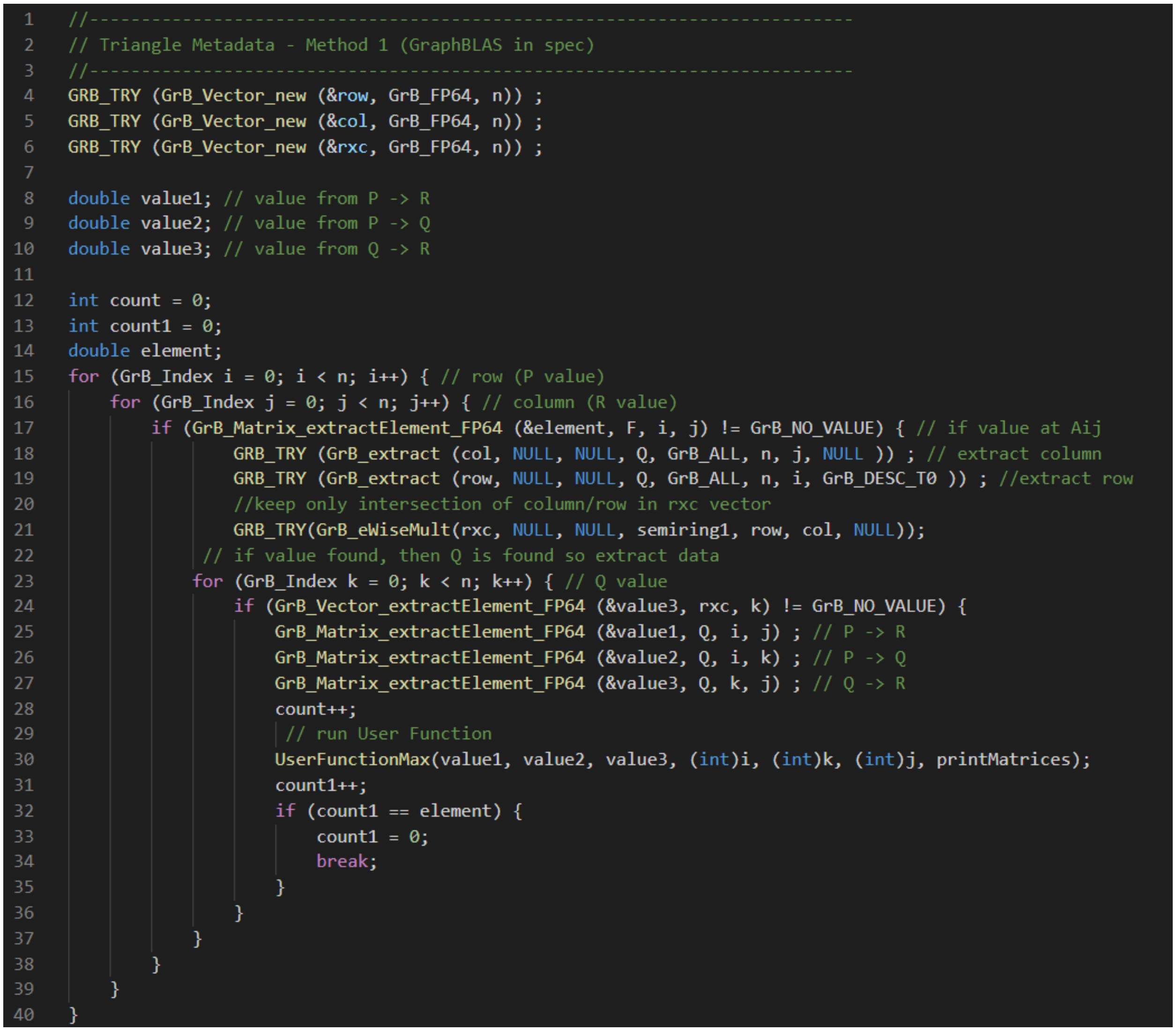


```
//---------------------------------------------------------------------------
// Triangle Metadata - Method 0 (GraphBLAS Iterators)
//---------------------------------------------------------------------------
// seek to A(0,:)
info1 = GxB_rowIterator_seekRow (iterator1, 0) ;
info2 = GxB_rowIterator_seekRow (iterator2, 0) ;
info3 = GxB_rowIterator_seekRow (iterator3, 0) ;

double v1, v2, v3;
while (info3 != GxB_EXHAUSTED) {
    // iterate over entries in A(i,:)
    GrB_Index i = GxB_rowIterator_getRowIndex (iterator3) ;

    // iterate over all columns corresponding to row i
    while (info3 == GrB_SUCCESS) {
        GrB_Index j = GxB_rowIterator_getColIndex (iterator3) ;

        // Move to A(i,*), and extract column index value
        info1 = GxB_rowIterator_seekRow(iterator1, i) ;
        GrB_Index j1 = GxB_rowIterator_getColIndex (iterator1) ;

        // Move to A(j,*), and extract column index value
        info2 = GxB_rowIterator_seekRow(iterator2, j) ;
        GrB_Index j2 = GxB_rowIterator_getColIndex (iterator2) ;

        while (info1 == GrB_SUCCESS && info2 == GrB_SUCCESS) {
            if (j1 > j2) {
                // A(i,j1) appears before A(j,j2)
                info2 = GxB_rowIterator_nextCol (iterator2) ;
                j2 = GxB_rowIterator_getColIndex (iterator2) ;
            } else if (j1 < j2) {
                // A(i,j1) appears after A(j,j2)
                info1 = GxB_rowIterator_nextCol (iterator1) ;
                j1 = GxB_rowIterator_getColIndex (iterator1) ;
            } else { // j2 == j1 => Triangle Found
                tcount++;
                // Get values of metadata
                v1 = GxB_Iterator_get_FP64 (iterator1);
                v2 = GxB_Iterator_get_FP64 (iterator2);
                v3 = GxB_Iterator_get_FP64 (iterator3);
                UserFunctionMax(v1, v2, v3, (int)i, (int)j, (int)j1, printMatrices);
                info2 = GxB_rowIterator_nextCol (iterator2) ;
                j2 = GxB_rowIterator_getColIndex (iterator2) ;
                info1 = GxB_rowIterator_nextCol (iterator1) ;
                j1 = GxB_rowIterator_getColIndex (iterator1) ;
            }
        }
        // move to next Column
        info3 = GxB_rowIterator_nextCol (iterator3);
    }
    // move to the next row, A(i+1,:)
    info3 = GxB_rowIterator_nextRow (iterator3) ;
}
```

Figure 2.4: Triangle Metadata Method 0 Code

2.1.4.3 Details for Method 1

Method 1 uses only GraphBLAS available in the spec to achieve the goal of finding the nodes of the triangles, and returning the metadata on the edges. The principle used in this Section to find the metadata comes from the output matrix of the Triangle Count. The Triangle Count output has a single edge of every triangle, and that edge is always the edge from P to R. Since the orientation of all the triangles is known due to the DODGr, this information can be used to find the missing node, and extract the data. Namely, due to the DODGr, there are 3 edges specifically in the order ((P, R), (P, Q), (Q, R)) corresponding to every triangle. Since the P and R nodes for the triangle are known, only the Q node need be found. In the DODGr matrix, the overlap of the outgoing nodes P, and the incoming nodes of R, is the value of the Q node. The outgoing edges of P are the row vector corresponding to P, and the incoming edges of R are the column vector corresponding to R. Everywhere in the triangle count output matrix where there is a value, the column and row vectors of the DODGr are extracted (for example, for the value at (1,2), row 1 and column 2 are extracted from the DODGr). Then, an eWiseMult (which finds the intersection of two vectors vectors) is applied to the column and row vectors, and wherever that intersection is, the missing Q value of the triangle is found. Then, the data can be extracted and given to the user so they can run the user defined function. The code for the algorithm is shown in Figure 2.5 below:

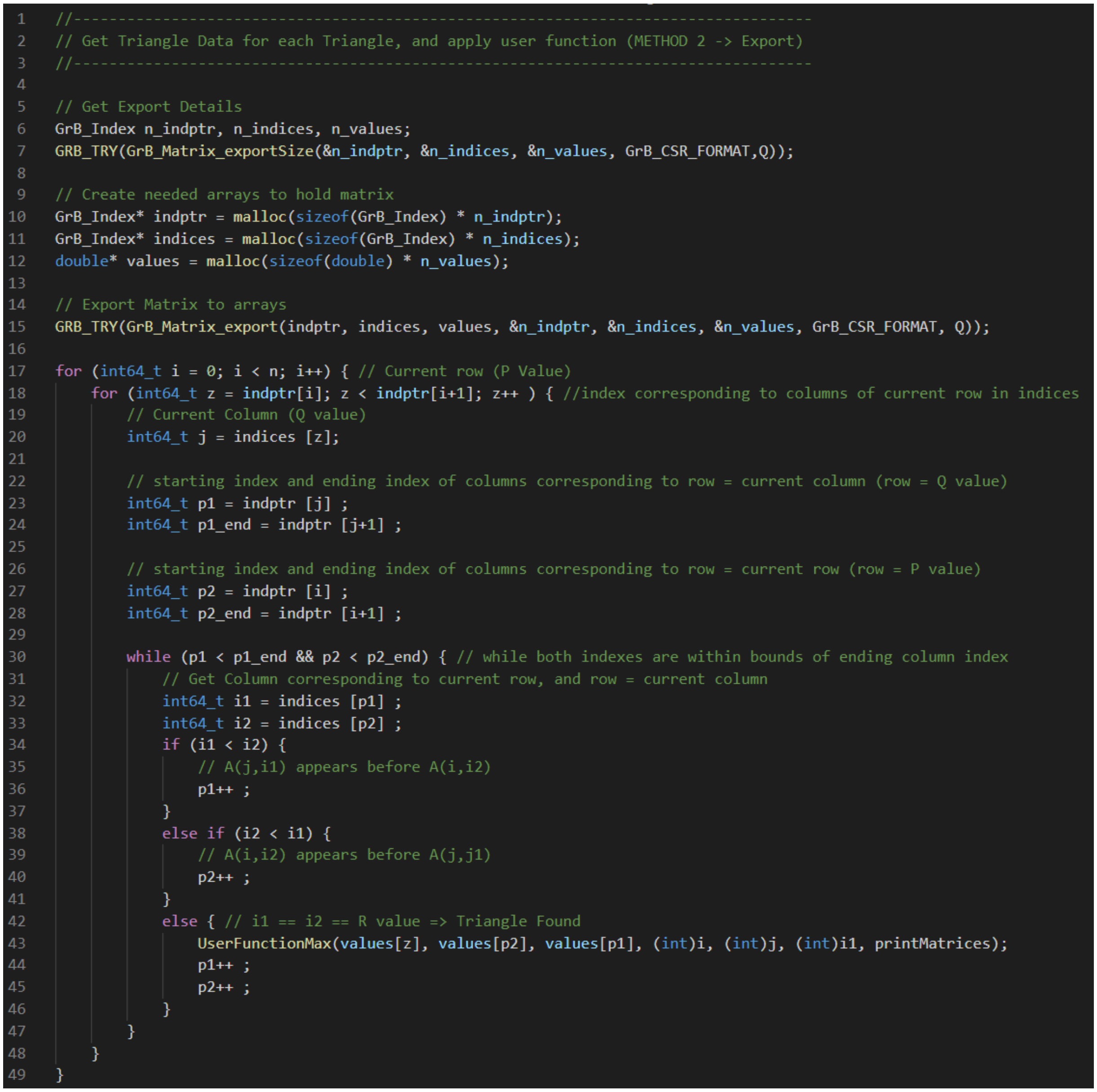


```c
//----------------------------------------------------------------------
// Triangle Metadata - Method 1 (GraphBLAS in spec)
//----------------------------------------------------------------------
GRB_TRY (GrB_Vector_new (&row, GrB_FP64, n)) ;
GRB_TRY (GrB_Vector_new (&col, GrB_FP64, n)) ;
GRB_TRY (GrB_Vector_new (&rxc, GrB_FP64, n)) ;

double value1; // value from P -> R
double value2; // value from P -> Q
double value3; // value from Q -> R

int count = 0;
int count1 = 0;
double element;
for (GrB_Index i = 0; i < n; i++) { // row (P value)
    for (GrB_Index j = 0; j < n; j++) { // column (R value)
        if (GrB_Matrix_extractElement_FP64 (&element, F, i, j) != GrB_NO_VALUE) { // if value at Aij
                GRB_TRY (GrB_extract (col, NULL, NULL, Q, GrB_ALL, n, j, NULL )) ; // extract column
                GRB_TRY (GrB_extract (row, NULL, NULL, Q, GrB_ALL, n, i, GrB_DESC_T0 )) ; //extract row
                //keep only intersection of column/row in rxc vector
                GRB_TRY(GrB_eWiseMult(rxc, NULL, NULL, semiring1, row, col, NULL));
             // if value found, then Q is found so extract data
            for (GrB_Index k = 0; k < n; k++) { // Q value
                if (GrB_Vector_extractElement_FP64 (&value3, rxc, k) != GrB_NO_VALUE) {
                    GrB_Matrix_extractElement_FP64 (&value1, Q, i, j) ; // P -> R
                    GrB_Matrix_extractElement_FP64 (&value2, Q, i, k) ; // P -> Q
                    GrB_Matrix_extractElement_FP64 (&value3, Q, k, j) ; // Q -> R
                    count++;
                     // run User Function
                    UserFunctionMax(value1, value2, value3, (int)i, (int)k, (int)j, printMatrices);
                    count1++;
                    if (count1 == element) {
                        count1 = 0;
                        break;
                    }
                }
            }
        }
    }
}
```

Figure 2.5: Triangle Metadata Method 1 Code

2.1.4.4 Details for Method 2

Method 2 extracts the DODGr matrix to multiple C arrays using the export command in GraphBLAS. Specifically, it exports the matrix in the CSR format. In the CSR format, there are three arrays used to represent the matrix: indptr, indices, and value. The indptr array is an array of size number of rows+1, where the index of this array represents the current row, and the value at the index represents the starting index in the indices array that correspond to the columns of the current row. The indices array is an array of numbers where each value represents the column to a corresponding row. Using indptr, the starting index and ending index of the columns corresponding to a row in the indices array can be found. The values array represents the value corresponding to the row and column index. The GraphBLAS C API [2] contains more information on the details of the export command and the CSR format. The idea is to use the previously found DODGr, and export it to the 3 arrays mentioned above. The code is shown in Figure 2.6 below:



```c
//-----------------------------------------------------------------------------------
// Get Triangle Data for each Triangle, and apply user function (METHOD 2 -> Export)
//-----------------------------------------------------------------------------------

// Get Export Details
GrB_Index n_indptr, n_indices, n_values;
GRB_TRY(GrB_Matrix_exportSize(&n_indptr, &n_indices, &n_values, GrB_CSR_FORMAT,Q));

// Create needed arrays to hold matrix
GrB_Index* indptr = malloc(sizeof(GrB_Index) * n_indptr);
GrB_Index* indices = malloc(sizeof(GrB_Index) * n_indices);
double* values = malloc(sizeof(double) * n_values);

// Export Matrix to arrays
GRB_TRY(GrB_Matrix_export(indptr, indices, values, &n_indptr, &n_indices, &n_values, GrB_CSR_FORMAT, Q));

for (int64_t i = 0; i < n; i++) { // Current row (P Value)
    for (int64_t z = indptr[i]; z < indptr[i+1]; z++ ) { //index corresponding to columns of current row in indices
        // Current Column (Q value)
        int64_t j = indices [z];

        // starting index and ending index of columns corresponding to row = current column (row = Q value)
        int64_t p1 = indptr [j] ;
        int64_t p1_end = indptr [j+1] ;

        // starting index and ending index of columns corresponding to row = current row (row = P value)
        int64_t p2 = indptr [i] ;
        int64_t p2_end = indptr [i+1] ;

        while (p1 < p1_end && p2 < p2_end) { // while both indexes are within bounds of ending column index
            // Get Column corresponding to current row, and row = current column
            int64_t i1 = indices [p1] ;
            int64_t i2 = indices [p2] ;
            if (i1 < i2) {
                // A(j,i1) appears before A(i,i2)
                p1++ ;
            }
            else if (i2 < i1) {
                // A(i,i2) appears before A(j,j1)
                p2++ ;
            }
            else { // i1 == i2 == R value => Triangle Found
                UserFunctionMax(values[z], values[p2], values[p1], (int)i, (int)j, (int)i1, printMatrices);
                p1++ ;
                p2++ ;
            }
        }
    }
}
```

Figure 2.6: Triangle Metadata Method 2 Code

As seen in Figure 2.6, the algorithm starts with a loop with n iterations, where n is the number of rows in the matrix. The next loop uses z=indptr[i] as a start, and z=indptr[i+1] as an end. Here, z represents the index for the indices arrays corresponding to the current column. So, z=indptr[i] to z=indptr[i+1] represents the index of the columns corresponding to row i. In the code, the variable j represents the actual column value corresponding to the index z (using the indices array and z as an index). The variable p1 represents the starting index of the columns corresponding to when the row is the current column (so the row j). p1_end is the ending index of those same columns. Variable p2 represents the starting index of the columns corresponding to the current row (which is row i). Variable p2_end represents the ending index of those same columns. In the while loop, the i1 and i2 represent the actual columns corresponding to the index's p1 and p2 respectively. At a high level, there is an edge from i to j, and also two other edges, one from j to i1, and the other from i to i2. If the nodes i1 and i2 were the same, then there would be a triangle ((i, j), (i, i1) (j, i1)), where the i-value corresponds to the P value, j corresponds to the Q value, and i1 (which equals i2 when there is a triangle) corresponds to the R value. This is precisely the function of the while loop in the code; it compares the value for i1 and i2, and (since the columns are in order) increases the lower of the two. Whenever they happen to be equal, there is a triangle, so the algorithm gives the metadata and the nodes to the user defined function.

3. RESULTS

3.1 Example Run of Algorithm

Here, an example of the algorithm running is shown, with outputs provided at every step. The input Graph to be used in the example run is shown in Figure 3.1. The circles represent Nodes, and the numbers on the edges represent metadata. The number of triangles that should be found is 2. For this run, the user function that will be run on the triangles is a simple max function, which will return the max value of the metadata of the edges, which should result in a value of 3 for the triangle with nodes (1,0,2), and a value of 6 for the triangle with nodes (3,2,4).

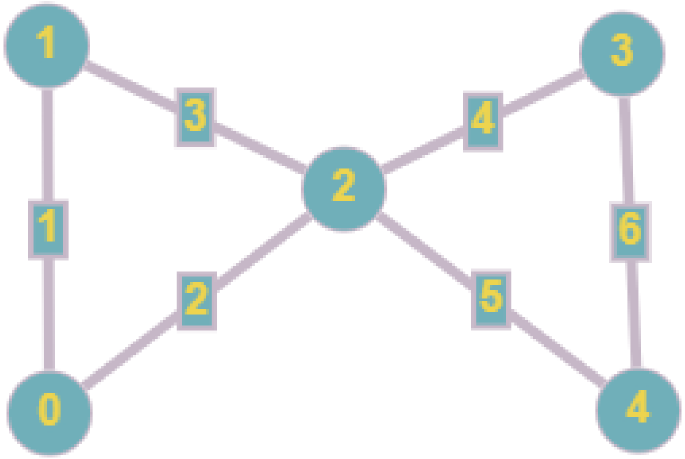
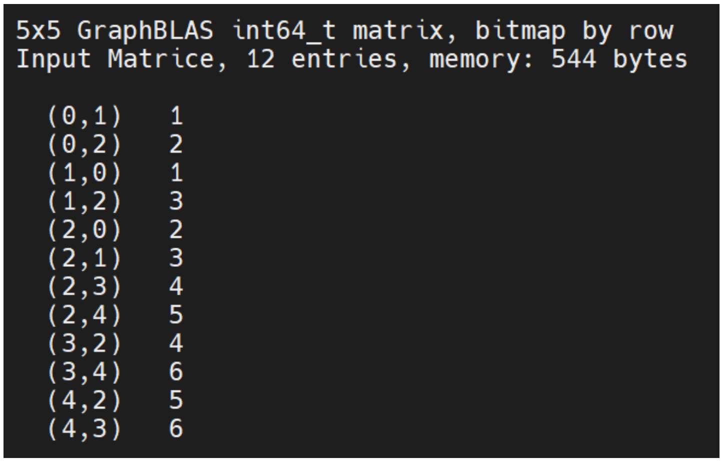


Figure 3.1: Input Graph used for Example Run

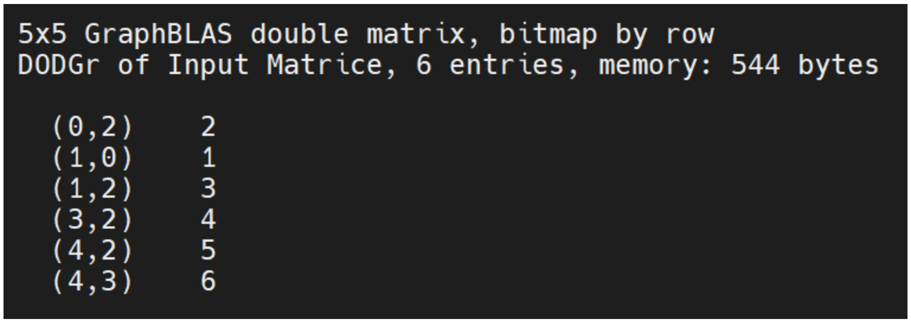
The first step to test is the Input Check. Here, if an unsupported Graph is given as input, then TriPoll in GraphBLAS outputs an error. For the purpose of this demonstration, the input will be assumed to be valid.

The second step is finding the DODGr of the input Graph. The input Graph as a matrix is shown in Figure 3.2. The matrix after applying the DODGr code (shown in Figure 2.2) is included in Figure 3.3. The matrix in Figure 3.3 is shown in the traditional Graph form in Figure 3.4. In Figures 3.2 and 3.3, (0,2) represents an edge from Node 0 to Node 2, and the number beside it (2 in this case) represents the metadata on that edge. As can be seen in Figure 3.4 and 3.3, the triangles are in the proper form for the DODGr, as described in Section 2.1.2, where nodes 1 = P, 0 = Q, and 2 = R for the first triangle, and nodes 4 = P, 3 = Q, 2 = R for the second triangle.



```
5x5 GraphBLAS int64_t matrix, bitmap by row
Input Matrice, 12 entries, memory: 544 bytes

  (0,1)   1
  (0,2)   2
  (1,0)   1
  (1,2)   3
  (2,0)   2
  (2,1)   3
  (2,3)   4
  (2,4)   5
  (3,2)   4
  (3,4)   6
  (4,2)   5
  (4,3)   6
```

Figure 3.2: Input Graph as a matrix



```
5x5 GraphBLAS double matrix, bitmap by row
DODGr of Input Matrice, 6 entries, memory: 544 bytes

  (0,2)    2
  (1,0)    1
  (1,2)    3
  (3,2)    4
  (4,2)    5
  (4,3)    6
```

Figure 3.3: Input Graph DODGr

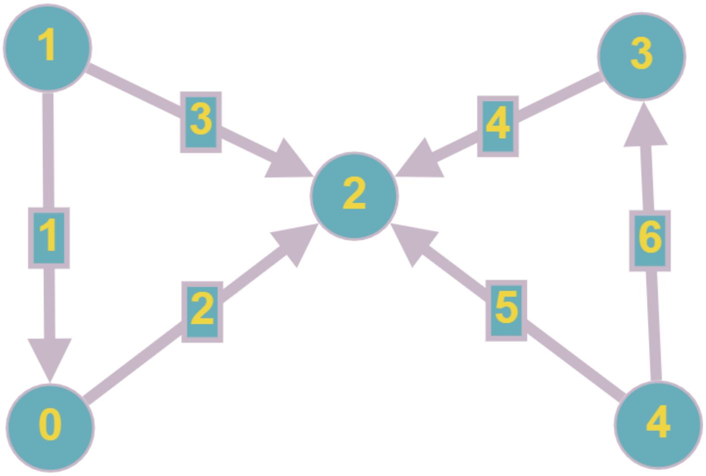
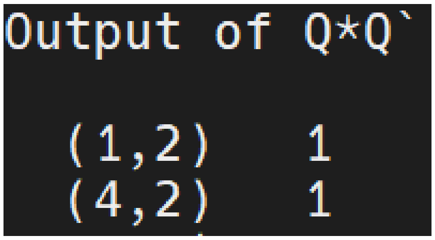


Figure 3.4: DODGr of Input Graph

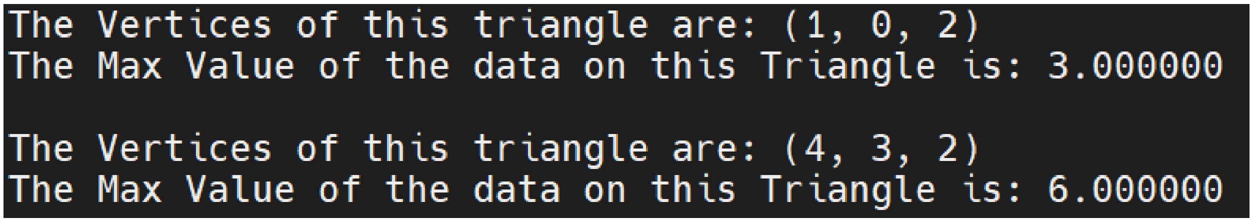
For the Triangle Count step, Figure 3.5 shows the output of running the code for Triangle Count shown in Figure 2.3. As mentioned in Section 2.1.3, summing the numbers in the output matrix should give you the number of triangles, which it does.



```
Output of Q*Q`

  (1,2)   1
  (4,2)   1
```

Figure 3.5: Output of Triangle Count step

For the Final Triangle Metadata Find step, Figure 3.6 shows the output of running the code for either of the three methods described in Section 2.1.4. The function, as mentioned above, simply outputs the nodes that make up the vertices of the triangle and the max value on the metadata on the edges. Of course, this is a simple example with simple numbers, but this can be run on Graphs with hundreds of thousands, or even millions, of triangles.



```
The Vertices of this triangle are: (1, 0, 2)
The Max Value of the data on this Triangle is: 3.000000

The Vertices of this triangle are: (4, 3, 2)
The Max Value of the data on this Triangle is: 6.000000
```

Figure 3.6: Output of Triangle Metadata Step

3.2 Performance Testing

The performance for of the three methods mentioned in Section 2.14 is shown below in Table 3.1. All three methods were run on the backslash TAMU server, which has a 12 core, 24 thread Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz, and 768GB of RAM. Each was run 10 times, and the average of the times was taken. The matrices being used are the bcsstk13.mtx file, and the cryg2500.mtx files included in LAGraph/data. The cvxbqp1.mtx and mawi_201512020000.mtx files from the SuiteSparse Matrix Collection were also included. Here, the printouts were disabled, as the number of triangles would overwhelm the terminal and be the limiting factor. The user defined function was the same as in the Example run (which was a simple max function). Also, time to read in Matrices were not included. An x means that the time was above 5 minutes, so the operation was canceled.

Table 3.1: Performance of TriPoll in GraphBLAS

	Jagmesh7.mtx (6,312 edges)	**bcsstk13.mtx (81,880 edges)**	**cvxbqp1.mtx (299,968 edges)**	**mawi_201512020000.mtx (74,485,420 edges)**
Method 0 Time (s)	.001859	.02376	.03004	5.321
Method 1 Time (s)	.1661	5.162	111.4	x
Method 2 Time (s)	.001846	.002006	.02817	5.074

4. CONCLUSION

4.1 Conclusions on Using GraphBLAS

GraphBLAS, and specifically the SuiteSparse: GraphBLAS, provides a powerful and expressive way to write Graph algorithms. However, as can be seen in the performance difference in the three methods in Section 3.2, the best way to use GraphBLAS is dependent on what the user is looking for. Method 1 uses only in-spec (following the GraphBLAS standard [2]) GraphBLAS to find the triangle metadata. In-spec GraphBLAS does not have an obvious way to find the exact nodes of each triangles using fast, bulk operations. Instead, it requires using single operations, which is not the main purpose of GraphBLAS and not where it excels in terms of performance. Method 1 was filled with single operations (specifically extracting elements), and because GraphBLAS stores its data opaquely, operations that seem like they would take constant time may be in fact taking logarithmic time or even greater. Although Method 1 is the slowest of the three (by a large amount), it is also the least complex. If the user is looking for the least complex option while still providing respectable performance, then Method 1 is a good option. Method 1 demonstrates that complexity can be largely reduced while keeping performance respectable, and may be useful if the user is less concerned with performance. However, when looking for better performance, Method 2 or Method 0 are recommended.

Method 2 exported the matrix into 3 arrays in CSR format, and found the nodes of the triangle that way. Method 2 still used GraphBLAS to find the DODGr of the Graph and the triangle count. It was only after these preliminaries were finished using GraphBLAS that the matrix was exported to finish the algorithm. This method represents another way GraphBLAS can be used for maximizing performance: using GraphBLAS where it can significantly reduce complexity while keeping performance high, and then exporting the matrix to finish the algorithm. The benefit of using this method is maintaining very high performance while still lowering complexity (in certain parts of the algorithm) by using GraphBLAS. The downside is that it is the most complex of the three methods.

Method 0 was recently added to the SuiteSparse implementation of GraphBLAS. It follows a similar pattern to Method 2, except that now the entire process takes place in GraphBLAS (in-place) using iterators. This allows for a less complex method that uses less space than method 2, all while keeping performance similar to Method 2. Method 0 shows the versatility and performance of GraphBLAS (and specifically SuiteSparse: GraphBLAS), where complexity is reduced, but performance is kept similar to Method 2 while keeping everything in GraphBLAS.

The goal of GraphBLAS is not necessarily to provide the best performance (although that may end up happening); rather, it is to provide strong performance while reducing complexity and keeping development time low. Method 1 is the least complex of the three methods, but is also noticeably slower than both Method 0 and Method 2. Method 2 is the fastest method, but is also the most difficult to both write and understand. Method 0 is a medium between the two methods. It is similar to Method 2 in that performance is very strong, but it is also similar to Method 1 in that it is purely in GraphBLAS, and it is easier to understand and write than Method 2.

Each of these Methods could be quite easily improved to increase performance through code optimization and parallelization. For example, the triangle counting code shown and used in this research is not the fastest, applying Wolf et al. [4] triangle counting code could be one of many ways to increase speed. That being said, the purpose of this research was not to maximize performance, but rather to illustrate what GraphBLAS can offer for different users when trying to balance complexity and speed. GraphBLAS, as can be seen through methods 0-2, has much to offer users no matter if what they are looking for is performance, low complexity, or even a middle ground.

REFERENCES

[1] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra.* Philadelphia, PA: SIAM, 2011.

[2] B. Brock, A. Buluc¸, T. Mattson, S. McMillan and J. Moreira. “The GraphBLAS C API specification,” http://graphblas.org/, Tech. Rep., 2021

[3] T. A. Davis, “Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and k-truss,” in *2018 IEEE High Performance extreme Computing Conference (HPEC),* Sep. 2018.

[4] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, “Fast linear algebra-based triangle counting with KokkosKernels,” in *2017 IEEE High Performance Extreme Computing Conference (HPEC),* Sept 2017, pp. 1–7.

[5] Steil, Trevor, Tahsin Reza, Keita Iwabuchi, Benjamin W. Priest, Geoffrey Sanders, and Roger Pearce. "TriPoll: computing surveys of triangles in massive-scale temporal graphs with metadata," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-12. 2021.