THE RICH CLUB COEFFICIENT AND NETWORK RANDOMIZATION WITH GRAPHBLAS

An Undergraduate Research Scholars Thesis

by

GABRIEL GOMEZ

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

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Timothy A. Davis

May 2025

Major: Computer Science

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, Gabriel Gomez, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Faculty Research 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

ABSTRACT

The Rich Club Coefficient and Network Randomization with GraphBLAS

Gabriel Gomez
Department of Computer Science and Engineering
Texas A&M University

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

This paper discusses the implementation of a new pair of algorithms built for network analysis using SuiteSparse: GraphBLAS, a linear algebra framework designed for streamlined implementation of sparse graph algorithms. Sparse graphs are networks where the number of edges within the graph is significantly lower than the possible number of edges; these networks are common in the real world, and their structures can be exploited for fast computations. This project will leverage the sparse matrix representation of graphs to implement two algorithms, one to calculate the rich club coefficient of a graph and another that randomizes a given graph to normalize the coefficient. The rich club coefficient measures the connectedness of high-degree nodes within a graph, which facilitates the analysis of network hierarchies. The new algorithm allows for the coefficients of a graph to be calculated thousands of times faster than the standard Python implementations of the algorithm. The rich club coefficient of a graph should be normalized by comparing it to a random graph with the same degree sequence. For this reason, this project will also focus on efficiently computing a random graph where every node maintains its degree. It implements an algorithm that recursively swaps edges in the graph to obtain a new graph with the same degree sequence. This provides a reliable and fast way to get new graphs with prescribed degree sequences.

ACKNOWLEDGMENTS

Contributors

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

Thank you to my fellow researchers Darin Peries, Hemanth Mukesh, and Casey Pei.

Thank you for anyone who took the time to listen to, talk about, or help edit my work. Including Dr. Oded Green, Dr. Francesca Marini, Darin Peries, Luigi Calabrese, Grant Lipham, Julio Dondisch, and Anika Sarna.

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

Finally, thank you to my parents for their encouragement.

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

Funding Sources

Undergraduate research was supported by Dr. Tim Davis at Texas A&M University.

1. INTRODUCTION

Graph algorithms written using linear algebraic operations have numerous advantages, providing them the potential to greatly improve efficiency and programming time. These algorithms can harness the power, efficiency, and parallelism of linear algebraic operations to solve complex problems more effectively. I implemented a novel algorithm to calculate the rich club coefficients (RCC) of a given graph and another to perform random edge swapping as a method of normalization for these coefficients.

1.1 Graph Algorithms in the Language of Linear Algebra

Graph algorithms are a cornerstone of network analysis, and they need to be efficient and quick to write. To this end, linear algebra has proven to be a powerful tool for solving graph problems in a highly parallelizable way. One way a graph can be represented as a matrix is with an adjacency matrix, A, where the entry in the i^{th} row and j^{th} column, denoted $a_{i,j}$, corresponds to an edge between nodes i and j. This representation and a variety of matrix operations allow for many algorithms to be made from these operations [1]. A graph and its representative matrix are considered sparse when relatively few of its possible edges exist. In real-world problems, these sparse networks are widespread, so the GraphBLAS C API Specification was designed to outline a library that would operate on these kinds of matrices. SuiteSparse: GraphBLAS (SSGB), developed by Dr. Tim Davis, is a full implementation of the GraphBLAS specification that aims to be as efficient and versatile as possible. The algorithms mentioned here, along with many others, are implemented within a GraphBLAS-based graph algorithm library named LAGraph.

1.1.1 The Matrix Semiring

A large part of what makes the matrix operations in GraphBLAS versatile is that the user can redefine some of the important operations within the matrix multiplication. A usual matrix multiplication consists of multiple dot products between row and column vectors; GraphBLAS allows the user to change how these dot products are calculated. To illustrate, the standard matrix multiplication, $C = AB$, is defined by:

$$c_{i,j} = \sum_{k=0}^{m} a_{i,k} \cdot b_{k,j} \tag{1}$$

GraphBLAS allows the programmer to construct a "semiring": a mathematical structure that consists of two binary operators on a set with certain properties. These two operators can replace the addition and multiplication operators that are used in a normal dot product. The mathematical definition for semirings is slightly more restrictive than the requirements for GraphBLAS, but this paper will focus on how GraphBLAS treats semirings. Matrix multiplication involving semirings will be denoted $A \oplus \otimes B$ for clarity. The $\oplus$ monoid replaces the usual addition function, while the $\otimes$ binary operation replaces multiplication. If $C = A \oplus \otimes B$ with $A \in D_a^{n \times r}$ and $B \in D_b^{r \times m}$ and $C \in D_c^{n \times m}$ then the following are required: [1, 2]

1. $\otimes : D_a \times D_b \to D_c$
2. $\oplus : D_c \times D_c \to D_c$
3. $a \oplus b = b \oplus a$ (the monoid is associative)

When working with sparse matrices, we assume that all empty elements, $\emptyset$, have the following properties for all x: $\emptyset \oplus x = x$ and $\emptyset \otimes x = \emptyset = x \otimes \emptyset$. This means that in a dot product, any element that multiplies with an empty results in an empty, and empties have no effect on addition. For the rest of this paper, empty elements will be denoted by a blank space. For example:

$$(a, \quad , c, d) \oplus \otimes (e, f, \quad , h)^T = ((a \otimes e) \oplus (d \otimes h)) \tag{2}$$

The following is an example of a common GraphBLAS semiring:

$$C = A \oplus \otimes B \qquad\qquad \oplus : (x, y) \mapsto x \text{ or } y \quad \otimes : (x, y) \mapsto 1 \tag{3}$$

This operation redefines both the standard addition and multiplication operators. First, $\oplus$ is an operator that will take any two non-empty inputs and output a 1. Additionally, $\oplus$ outputs either of the values it is given. Note that this normally isn't an associative operation, but since its inputs are only 1's, it is associative. So, this semiring would yield a 1 in the $c_{i,j}$ place when the i^{th} row in A and j^{th} column in B have any overlap. Otherwise, that entry will be empty. GraphBLAS also allows for masks and accumulators for any operation. The mask, denoted $C\langle M\rangle = A \oplus \otimes B$, allows the user to restrict what parts of the array can be modified by a matrix operation. The accumulator, denoted $C {+}{=}\ A \oplus \otimes B$, adds the results back into an existing matrix; this addition operation can be replaced by any binary operator.

Some other basic operations in GraphBLAS include element-wise multiplication, which is denoted $C = A \otimes B$. This takes two matrices of the same dimension and directly pairs their entries, outputting the result into the corresponding entry in matrix C. However, if the value in A or B does not exist, then no value will be added to C. Element-wise addition $C = A \oplus B$. works identically, except that if the value in A or B does not exist, then C gets the value of the entry that does exist. If neither exists, then no value is added to C.

1.2 The Rich Club Coefficient

Research on the rich club coefficient (RCC) has revealed the existence of a "rich club phenomenon" within many networks. That is, in many networks, higher-degree nodes preferentially connect. The RCC of a graph for a given number k is calculated as the ratio of edges between nodes of degree higher than k, to the total possible number of edges between nodes of degree higher than k. Specifically:

$$\phi(k) = \frac{2E_{>k}}{N_{>k}(N_{>k} - 1)} \tag{4}$$

where $E_{>k}$ is the number of edges among nodes of degree higher than k and $N_{>k}$ is the number of nodes of degree higher than k. [3]

1.3 Graph Randomization

One important point about the RCC is that it is natural for higher-degree nodes to be more interconnected, even within a random network. So, when studying the "rich club" of networks, it is important to normalize the RCC to account for this probabilistic tendency. One such method of normalization is to compare the RCC of a random graph where each node has the same degree as those in the graph we mean to study. In a given graph, if there are two edges without the same vertices, then pairing the edges and swapping one of the vertices of the first with a vertex of the second makes a new slightly different graph where each vertex has maintained its degree. This operation is referred to as edge swapping.

$$\text{edge\_swap}((a, b), (c, d)) \mapsto ((a, c), (b, d)) \text{ OR } ((a, d), (b, c)) \tag{5}$$

The goal of this project is to write an algorithm that will do thousands or even millions of these swaps in parallel (depending on the problem size) and enable the creation of large random graphs that can be compared to other graphs to facilitate their analysis.

1.3.1 The Incidence Matrix

When attempting to do operations on the edges of a graph, it is sometimes difficult to work with the adjacency matrix. There is a different type of matrix that gives finer access to every edge in the graph. This matrix, denoted E, is called the incidence matrix, where instead of having one vertex per row and one vertex per column, it has each edge represented as a row, and each column representing a vertex. Then, each row in a standard graph has two entries, representing the two vertices that are connected by that edge. LAGraph has an algorithm to create this incidence matrix, contributed by Madhu [4]. Using the incidence matrix allows for the easier formulation of the edge-swapping algorithm.

2. METHODS

2.1 The Rich Club Coefficients

2.1.1 Overview

The RCC was calculated as shown in **Equation 4** for a given k by calculating $E_{>k}$, the number of connections among nodes bigger than k, and $N_{>k}$, the number of nodes with degree greater than k. `LAGraph_RichClubCoefficient` calculated the RCCs for all $k < d_{max}$ of a graph where d_{max} is the highest degree of the graph. To do this, we first calculated the minimum degree of the two nodes belonging to each edge, because if both nodes are of a degree greater than k then the edge will count towards the total in $E_{>k}$. This was done by first calculating the degree vector, which can be a cashed property of a graph in LAGraph and is calculated as follows:

$$\mathbf{d} = A \oplus \otimes \mathbf{1} \qquad \oplus : (x, y) \mapsto x + y \quad \otimes : (x, y) \mapsto 1 \tag{6}$$

where $\mathbf{1}$ is a dense vector of all ones, and A is the adjacency matrix. This multiplication works by finding all of the present entries in A for a given row and assigning them a value of 1. Then, it sums all of the ones, which yields the number of entries in the row. Since these entries are equivalent to connections on a graph, this sum is the out-degree of the vector. With this vector, we can make a diagonal matrix D with the diagonal entries having the same values as $\mathbf{d}$. Then, to calculate the minimum degree of every edge, we do the following:

$$A_d = D \oplus \otimes A \qquad \oplus\, (x, y) \mapsto x + y \quad \otimes : (x, y) \mapsto x \tag{7}$$

$$A_{min} = A_d \oplus \otimes D \qquad \oplus\, (x, y) \mapsto x + y \quad \otimes : (x, y) \mapsto \min(x, y) \tag{8}$$

Then, $E_{>k}$ can be found simply by counting up all of the entries in the matrix whose value is greater than k and $N_{>k}$ by counting how many entries in the $\mathbf{d}$ vector values greater than k. After that, finding the RCC is straightforward using **Equation 4**.

2.1.2 Implementation

Algorithm 1 Rich Club Coefficient Pseudo Code in Linear Algebra

```
Input: $G$: A graph with adjacency matrix $A$
Input: $G.out\_degree$: The cached degree sequence of $G$
Vector $\mathbf{d} = G.out\_degree \oplus -\mathbf{1}$    ▷ $\oplus \equiv +$. Subtract by 1 and fill empties with -1
Matrix $D \leftarrow \text{diag}(\mathbf{d})$
Matrix $A_d = D \oplus \otimes A$    ▷ $\oplus : (x, y) \mapsto x \quad \otimes : (x, y) \to x$
Vector $\mathbf{e_{vert}} = A_d \oplus \otimes \mathbf{d}$    ▷ $\oplus : (x, y) \mapsto x + y \quad \otimes : (x, y) \to 2(x < y) + (x = y)$
$\mathbf{d} = \text{Select}(\mathbf{d} : x \geq 0)$    ▷ remove the $-1$'s that were added to deg
$(\mathbf{i}, \mathbf{v}) = \mathbf{d}$
$(\mathbf{i}, \mathbf{u}) = \mathbf{e_{vert}}$
Vector $\mathbf{e_{tot}} = (\mathbf{v}, \mathbf{u})$    ▷ Duplicate numbers in $\mathbf{v}$ resolved by summing $\mathbf{u}$ values.
Vector $\mathbf{d_{tot}} = (\mathbf{v}, \mathbf{1})$    ▷ Duplicate numbers in $\mathbf{v}$ resolved by counting
$\mathbf{e_{tot}} = \text{inv\_cum\_sum}(\mathbf{e_{tot}})$
$\mathbf{d_{tot}} = \text{inv\_cum\_sum}(\mathbf{d_{tot}})$
return $\mathbf{d_{tot}} \otimes \mathbf{e_{tot}}$    ▷ $\otimes : (x, y) \mapsto \frac{x}{y(y-1)}$
```

Algorithm 1 displays the implementation of `LAGraph_RichClubCoefficient`. It has a few more optimizations than the ideas in the previous section. First, it only does one multiplication involving the D matrix, which assigns each row's values to the number of entries in that row (ie, the degree of that row). This operation was optimized by filling out the diagonal of the D matrix with -1. The -1s on the diagonal are not multiplied by anything because they correspond to empty rows in the matrix; however, a full, diagonal, left-matrix multiplication signals to SSGB that it only needs to move down the rows of matrix A and change that row's values depending on its row number. This leads to SSGB using a much quicker method for that operation.

The next matrix-vector multiplication effectively counts the edges for each node. However, it assigns 0 to edges for which the vertex corresponding to its row has the greater degree, 1 if the row and column have the same degree, and 2 if the row vertex has the lowest degree. Essentially, this ensures that each edge gets counted exactly twice and that it corresponds to its lowest degree vertex. Here is an example of multiplication with this semiring:

$$\begin{pmatrix} & 2 & & & 2 \\ 2 & & & 2 & \\ & & & 2 & 2 \\ & 3 & 3 & & 3 \\ 3 & & 3 & 3 & \end{pmatrix} \oplus \otimes \begin{pmatrix} 2 \\ 2 \\ 2 \\ 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 1+2 \\ 1+2 \\ 2+2 \\ 0+0+1 \\ 0+0+1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \\ 4 \\ 1 \\ 1 \end{pmatrix} \tag{9}$$

Next, the program makes two vectors with the desired values: $E_{>K}$ and $N_{>k}$. It does this by building a vector with the degree of each vertex as the index of the edges that were counted for that vertex. It resolves duplicated indices by adding the values of the counted edges together. Originally, this involved a call to `GrB_Vector_Build`. However, it was later replaced with a call to `LAGraph_FastAssign`, a utility function created for the edge-swapping algorithm that will be explained later in this paper. One more vector counts the number of vertices with a certain degree. This is made by using degrees as indexes and 1's as values and resolving duplicates with addition. Once the $\mathbf{e_{tot}}$ and $\mathbf{n_{tot}}$ vectors are made, the cumulative sum starting from the highest degree makes $\mathbf{e_{tot}}[k] = E_{>k}$ and $\mathbf{n_{tot}}[k] = N_{k>}$. Finally, to find the RCC, the algorithm applied the RCC formula from Equation 4 to each of the entries of the two vectors using an element-wise multiply.

2.2 Graph Randomization

2.2.1 Overview

An edge swapping process performed the graph randomization for this project [5]. In this process, we found any two edges in an undirected graph and swapped their vertices as long as the swap did not create a self or parallel edge. The edges (a, b) and (c, d) could be changed into (a, c) and (b, d), or (a, d) and (b, c). A sample of this swapping is demonstrated in Figure 1.

Since SSGB is a parallel library, the goal was to make many of these swaps in parallel. To accomplish this, we first paired a large majority of the edges together to know which edges would

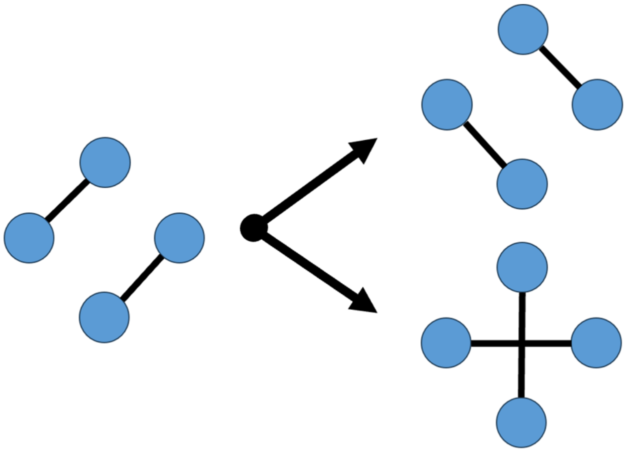

Figure 1: Edge Swapping Options

swap by getting a partial permutation of the edges using a hashing method inspired by Green and colleagues [6]. This hashing method also checked if the newly created edges had any duplicates in the existing graph since a swap should not be done if it would create a parallel edge.

2.2.2 The Fast Assign Method

2.2.2.1 Purpose

Throughout the process of edge swapping and calculating the RCC, one costly procedure was placing a large number of values at unordered indices and applying an operator to duplicates at those indices. GraphBLAS has a method for doing this called `GrB_Vector_Build`, but this method would often take too long, as it first sorts the indices given to it before building the vector. The range of indices was often tight enough that it would make more sense to place the value of that point directly in the vector using random access into an array. This type of build, however, has not yet been implemented into SSGB. Another algorithm that could have been useful for this is `GrB_Vector_Assign`; however, assign does not have the flexibility to apply an operator to duplicate entries, and the algorithm built here runs about three times faster.

Since these builds comprised a majority of the runtime for the edge-swapping algorithm, it was pertinent to find a quicker solution than `GrB_Vector_Build`. Essentially, I was looking for an algorithm to take advantage of random access into a vector to place values at unsorted indices.

This would be equivalent to what is shown in Algorithm 2. This loop is not parallel, and it also does not use the Suitesparse: GraphBLAS library. I aimed to find a method that would make GraphBLAS do this same operation for me.

Algorithm 2 Random Access Indexing Algorithm

```
Input: $\mathbf{c}$: The output vector: may contain values already.
Input: $\mathbf{i}$: A full vector of indices for values to be placed at
Input: $\mathbf{x}$: A vector of values to be placed at given indices
Input: $\oplus$: a monoid to be applied to duplicate values on the same index
for $j = 0$ to len($\mathbf{i}$) do
    $k = \mathbf{i}[j]$
    $\mathbf{c}[k] \oplus = \mathbf{x}[j]$
end for
```

2.2.2.2 Implementation

There is a method that has been used in the connected components algorithm of LAGraph. It involves a similar case to the above, where the developers wanted to place values at unsorted indices while applying a duplicate monoid. To do this, they made a matrix using a historical method of SSGB called `GxB_Matrix_Pack_CSC`, which used move semantics to build a matrix in constant time if the right data is passed in. This method was used throughout my algorithm, and a new version of SuiteSparse created a different method for creating matrices using move semantics. So, the best solution was to move this trick into its own function and code it to be compatible with SSGB Version 10.

The idea behind the `LAGraph_FastAssign` utility is to create a matrix P such that each column j of P has a singular entry at row $\mathbf{i}[j]$. The value of this entry is irrelevant. So, each column j of P has a value placed at exactly the row where $\mathbf{c}$ should get the value of $\mathbf{x}[j]$. Therefore, the dot product between that entire row and $\mathbf{x}$ will apply the multiplication operator anywhere where $\mathbf{i}$ has the value for that row. The multiplication operator takes two values and returns the second (known as the SECOND operator in GraphBLAS), resulting in each intersection on the row getting the j^{th} value from the $\mathbf{x}$ vector, and they will be combined using the provided duplicates monoid. Algorithm 3 shows how this operation works.

Algorithm 3 Fast Assign Pseudo Code

```
Output: Vector $\mathbf{c}$: The output vector: may contain values already.
Input: Vector $\mathbf{i}$: A vector of indices for values to be placed at
Input: Vector $\mathbf{x}$: A vector of values to be placed at given indices
Input: Vector $\mathbf{r}$ (optional): A prebuilt vector $\mathbf{r} = [0, 1, 2, \ldots, \text{len}(\mathbf{i}) - 1]$
Input: $\oplus$: a monoid to be applied to duplicate values on the same index
Vector $\mathbf{r} = [0 : \text{len}(\mathbf{i}) - 1]$                ▷ If $\mathbf{r}$ was not passed as input, build it.
Matrix $P = (\mathbf{i}, \mathbf{r}, \mathbf{1})$                    ▷ This can be done in O(1) time through a load operation
$\mathbf{c} \mathrel{+}= P \oplus \otimes \mathbf{x}$                ▷ $\otimes : (x, y) \mapsto y$
```

The construction of P can be done in constant time because of an operation in SSGB that uses move semantics to receive a matrix from the user or give the matrix to a user. In the newest version of SSGB, this operation is called load/unload. It allows the user access to a non-opaque GxB_container type with all of the internal data of the matrix. SSGB uses a variety of sparse matrix representations; the one used for this method was the compressed sparse column (CSC) format. CSC matrices are represented by 3 vectors so that this matrix,

$$A = \begin{bmatrix} 4 & & 3 & \\ 3 & 2 & & 1 \\ & 1 & 3 & \\ 3 & 1 & & 1 \end{bmatrix} \tag{10}$$

is held in CSC form as follows:

```
integer Ap [ ] = { 0,       3,       6,    8,   10 } ;
integer Ai [ ] = { 0, 1, 3, 1, 2, 3, 0, 2, 1, 3 } ;
integer Ax [ ] = { 4, 3, 3, 2, 1, 1, 3, 3, 1, 1 } ;
```

Where Ap$[i]$ holds the starting point for the i^{th} column, Ai holds the indices that exist within that column, and Ax holds the matrix values at those indices. With this format, it is simple to describe the matrix P. If $\mathbf{x}$ has n entries, then P must have n columns. Since each column only has one value, Ap = $[0, 1, \ldots, n]$. Ai contains the index of the only row that is present in each column, which are the desired indices to place the values at ($\mathbf{i}$), and Ax would be an array of all ones. Because of a special feature of SSGB, a full array that has a singular value can be built and stored with constant time and space. Ai would already be constructed because it is the same as the input vector, $\mathbf{i}$, containing the desired indices. The only vector that needs to be constructed is Ap, however, this vector does not change and can be reused, so FastAssign accepts a preconstructed vector that can be used to make the construction of P constant time.

This utility is useful because it can be much faster than other SSGB methods, as shown in the results section in **Table 3**. It is also much more flexible than assign because it can apply a duplicate monoid to duplicate entries in the vector. When compared to `GrB_Build`, FastAssign is often faster, and it can work on a vector that already has entries in it. One drawback of FastAssign is that it does not have the well ordering of the `GrB_Build` duplicate operator. The SSGB version of `GrB_Build` allows for the duplicate operator to be non-commutative and guarantees that the first value assigned to an index will be the first variable in the duplicate operator. This can certainly be useful in some cases but was not a large consideration for FastAssign's use in the swap edges algorithm. The other drawback is that the kernels being targeted by FastAssign can be quite fragile, so the programmer using this utility has to choose the data being input carefully to ensure a fast FastAssign.

Specifically, since the main goal of FastAssign is to get SSGB to use random access into vectors, the output vector $\mathbf{c}$ must be of a format that allows random access. That means it should either be bitmap or full. This can be set using the command `GrB_set`. In the edge-swapping code, this command is often used, specifically to help the performance of FastAssign. Additionally, if an accumulator is used, and that accumulator is the same operation as the $\oplus$ moniod being used for FastAssign, SSGB has a fast kernel (saxpy4/5) that currently only works if the output vector $\mathbf{c}$ is full. So, the `LAGraph_SwapEdges` code often filled vectors that could have been sparse to help the runtime of the algorithm.

2.2.3 *Implementation*

As mentioned before, this algorithm uses a structure similar to the incidence matrix E. It is essentially a vector of pairs ($\mathbf{E}$), with each entry representing an edge between a pair of vertices in the graph. This vector can be constructed by getting the indices of the entries in the adjacency matrix $(\mathbf{i_A}, \mathbf{j_A}) = A$, then $\mathbf{i}$ is the first column of $\mathbf{E}$ and $\mathbf{j}$ is the second. When the edges were paired for swapping, they were put into a vector that was half the length and held a tuple of four integers representing the vertices involved in the swap.

Algorithm 4 shows the process for parallel edge swapping. The initial permutation is created on line 12 by placing edges at random indices. Collisions were resolved with an operator that would randomly choose between the numbers to ensure a random sampling in the permutation. Note that this leads to some edges in E not appearing in $\mathbf{m}$, which is not a problem because each loop swaps around $\frac{2}{3}$ of the edges at a time. Adjacent pairs of edges in the permuted vector were then swapped, with the type of swap decided by their placement in the permuted vector. This swap was accomplished by putting each pair of edges into an entry in the vector $\mathbf{M}$, which is essentially a vector of 4-tuples. This operation was constant time because $\mathbf{m}$ and $\mathbf{M}$ are both full vectors with the same data; the only difference is how the type of the data is interpreted. So, using the move semantics of load/unload, SSGB can switch its interpretation of the data type quickly. This representation simplified pairing edges together and swapping the vertices within two edges using `GrB_apply`, which applies a function onto as vector, in this case the edge_swap function seen in **Equation 5**. Then, a hashing algorithm was used to assign each edge a hash with a range of $\sim 4e$. This allowed for duplicate edges to be found quickly using the fast assign method with a relatively low risk of collision. The SSGB kernels I target with FastAssign are somewhat sensitive, so measures were taken to ensure performance. For example, while $\mathbf{s_e}$ could be a sparse vector, making it full led to a much quicker SSGB kernel being used and a $4\times$ speed-up in the operation on line 21 for some graphs. The duplicate operation on line 21 is not addition, this is because the

Algorithm 4 Edge Swapping Pseudo Code

```
Input: Graph $G$: An LAGraph graph to be randomized
Input: $Q$: Desired swaps per edge
Matricies $A$, $A_{tril}$
Vectors $\mathbf{E}$, $\mathbf{s}$, $\mathbf{r}$, $\mathbf{r_{perm}}$, $\mathbf{m}$, $\mathbf{M}$, $\mathbf{h_E}$, $\mathbf{h_m}$, $\mathbf{s_e}$, $\mathbf{m_{val}}$
$A \leftarrow$ the adjacency matrix of $G$.
$A_{tril} \leftarrow a_{i,j} \in A$ such that $j < i$        ▷ AKA the lower triangular part of the matrix.
$e \leftarrow$ number of edges in $A_{tril}$
$n_{swap} \leftarrow 0$
$(\mathbf{i}_A, \mathbf{j}_A) = A_{tril}$
$\mathbf{E}[i] = (\mathbf{i}_A[i], \mathbf{j}_A[i])$        ▷ Each entry is an edge represented by a tuple
$\mathbf{s} = [0 : e - 1]$
while $n_{swap} < e * Q$ do
    $\mathbf{r} \leftarrow$ random values $0 \leq \mathbf{r}[i] \leq 2e$. len($\mathbf{r}$) $= e$.
    $\mathbf{r_{perm}} = (\mathbf{r}, \mathbf{s})$        ▷ Using FastAssign, get a partial permutation of numbers 0 to $e$
    $\mathbf{r_{perm}} \leftarrow$ values of $\mathbf{r}_{perm}$        ▷ Place the permuted values in a dense vector
    $\mathbf{m}[i] = \mathbf{E}[\mathbf{r}_{prem}[i]]$        ▷ Place the edges into $\mathbf{m}$ using the order dictated by $\mathbf{r}_{perm}$
    $\mathbf{M}[i] = (\mathbf{m}[2i], \mathbf{m}[2i + 1])$        ▷ Put adjacent edges in the same entry in a new vector
    $\mathbf{M} = \text{swap}(\mathbf{M})$        ▷ swaps two edges as mentioned in Figure 1
    $\mathbf{m}[i] = \mathbf{M}[i/2][i\%2]$        ▷ Put the new edges back into a one edge per entry vector.
    $\mathbf{h_E}[i] = \text{hash}(\mathbf{E}[i])$
    $\mathbf{h_m}[i] = \text{hash}(\mathbf{m}[i])$
    $\mathbf{s_e} = (\mathbf{h_E}, \mathbf{1})$        ▷ Using FastAssign, place a one at any hash value from an edge in $E$
    $\mathbf{s_e} += (\mathbf{h_m}, \mathbf{1})$        ▷ Using FastAssign, add up the hashes of $\mathbf{m}$ that fit into each bucket
    From $\mathbf{s_e}$ select $\mathbf{s_e}[i]$ such that $\mathbf{s_e}[i] = 1$        ▷ Only select hash buckets with no collisions
    $\mathbf{m_{val}}[i] = \mathbf{s_e}[\mathbf{h_m}[i]]$        ▷ Done via a vector extract
    $\mathbf{m_{val}}[i] = \mathbf{m_{val}}[i]$ land $m_{val}[i$ bxor $1]$        ▷ New edge is valid if both edges in the swap are
    $\mathbf{m}\langle\mathbf{m_{val}}\rangle = \mathbf{m}$        ▷ Remove invalid edges from $\mathbf{m}$
    $\mathbf{E}[\mathbf{r}_{prem}[i]] = \mathbf{m}[i]$        ▷ Using FastAssign, place valid edges back into $\mathbf{E}$
    $n_{swap} += \text{nvals}(\mathbf{m_{val}})$
end while
$(\mathbf{i}_A[i], \mathbf{j}_A[i]) = \mathbf{E}[i]$
$A_{tril} = (\mathbf{i}_A, \mathbf{j}_A)$
return $A_{tril} + A_{tril}^T$
```

only information that matters is if there is more than one entry in a hash bucket of $\mathbf{s_e}$, and $\mathbf{s_e}$ is an 8-bit integer vector. So, using addition as the duplicate operator could result in overflows. Instead, I use an operator $\oplus : (x, y) \mapsto \min(x + y, 2)$. Line 24 uses a similar method as line 16 to do the operation with an in-place matrix apply. This operation was necessary because if one of the edges that swapped had to be reversed (because it was parallel or a self edge) then the edge that it swapped with could not change either. The masked assignment on line 25 was done without using an assign operation by asking SSGB to use the data from the $\mathbf{m_{val}}$ vector as the bitmask for the $\mathbf{m}$ vector. A bitmask is essentially a Boolean data structure in GraphBLAS that indicates whether or not an entry in a C array is present in the vector. By replacing the bitmask of $\mathbf{m}$ with $\mathbf{m_{val}}$, SSGB will treat the 0's in $\mathbf{m_{val}}$ as empty values in $\mathbf{m}$. So, when line 26 places the new edges back into the $\mathbf{E}$ vector, swaps that were not valid will not be placed into $\mathbf{E}$.

3. RESULTS

3.1 The New Algorithms

There are now two new algorithms and a utility method, with tests and benchmarking that have been implemented into the LAGraph library. `LAGraph_RichClubCoefficient` takes a graph and returns a vector with its RCCs for every k. This method is thousands of times faster than its NetworkX equivalent on small graphs and is much more practical for larger graphs. `LAGraph_FastAssign` can be used in a variety of LAGraph algorithms, and it was partially inspired by a similar method in one of LAGraph's connected components methods. So, the code for that algorithm was significantly simplified while maintaining its performance by using `LAGraph_FastAssign`. FastAssign was also incredibly useful in the implementation and simplification of the `LAGraph_SwapEdges` code. The swap edges algorithm relied on many random access operations to function, which was initially difficult to think of linear-algebraically, but the FastAssign algorithm and its variety of use cases displayed in SwapEdges will inspire future uses of GraphBLAS. The source code for all of these functions can be found in the appendix or their most recent versions on the LAGraph GitHub.

3.2 Tables and Figures

Table 1 shows the sizes of the matrices that were used to benchmark the algorithms. The first five of these come from the GAP benchmark, and the last three from the SNAP collection [7, 8]. Benchmarking was done on the `Backslash` computer at Texas A&M, which boasts an Intel Xeon E5-2695 v2 processor running at 2.40GHz with 12 cores, one socket, 24 threads, and 768 gigabytes of memory. The results of this benchmark are shown in **Table 2** for the RCC algorithm and **Table 4** for the edge swapping algorithm. Where possible, the NetworkX times for the same graph are shown for reference. These results indicate that the algorithms effectively used the threads they were allocated, especially in the edge swapping algorithm, which demonstrates up to a $20\times$ speed up when running with twenty-four cores rather than one.

Table 1: Dimensions of the Banchmarking Graphs

Name	Edges	Vertices
Kron	4,223,264,644	134,217,726
Twitter	2,405,026,092	61,578,415
Road	57,708,624	23,947,347
Urand	4,294,966,740	134,217,728
Web	3,620,126,660	50,636,151
Youtube	5,975,248	1,134,890
LiveJournal	69,362,378	3,997,962
Orkut	234,370,166	3,072,441

The RCC code demonstrated decent parallelism with an average speed up of $8\times$ on the GAP benchmark graphs, making the efficiency of the thread use 34%. It runs quickly, even on large graphs, although its performance can be heavily dependent on graph structure. When compared to the NetworkX algorithms, the LAGraph algorithm far outperforms the NetworkX implementation, which is expected since NetworkX is a python library and the code is single threaded. This new algorithm will be useful to researchers who need the power to compute RCCs on large graphs quickly since it is the first implementation of RCC in a C library.

Table 2: Times for RCC

Name	24 Threads	8 Threads	3 Threads	1 Thread	NetworkX
Kron	14.110 s	30.927 s	58.215 s	172.442 s	-
Twitter	7.840 s	10.245 s	19.748 s	51.079 s	-
Road	0.326 s	0.392 s	0.754 s	2.016 s	579.838 s
Urand	17.455 s	23.909 s	59.948 s	180.083 s	-
Web	7.038 s	7.495 s	12.027 s	43.189 s	-
Youtube	0.035 s	0.027 s	0.059 s	0.142 s	64.284 s
LiveJournal	0.210 s	0.244 s	0.369 s	1.106 s	732.994 s
Orkut	0.778 s	0.583 s	2.040 s	3.690 s	2449.956 s

Table 3 shows the speed of `LAGraph_FastAssign` when placing 1s at random indexes in a vector. It is compared to some GrB methods that do the same operation, and a single threaded

for loop. This utility function also maintains this speed while allowing for duplicate handling. This means it is more flexible than `GrB_Assign`, which does not do duplicate handling, and becomes much faster than `GrB_Build`, which slows down significantly when it has to handle duplicates.

Table 3: Times to Index Random Hashes into Buckets

Number of Hashes	FastAssign (24)	GrB_Assign (24)	GrB_Build (24)	For Loop (1)
100,000,000	0.449 s	1.221 s	0.778 s	2.270 s
1,000,000,000	4.833 s	13.680 s	8.927 s	35.249 s
10,000,000,000	66.312 s	251.279 s	133.477 s	853.056 s

In **Table 4**, the times for edge swapping are shown for one swap per edge, although the recommended number of swaps per edge is 100 [5]. This algorithm is one of the costliest algorithms in LAGraph, but it still far outperforms NetworkX's algorithm, even running on a single thread. This new algorithm makes edge swapping viable for much larger graphs than previously possible since, for example, GAP-Twitter was randomized at 100 swaps per edge in 4 hours and 40 minutes. The final run time for this algorithm is significantly lower when compared to its starting point. This is in large part thanks to the efficacy of the FastAssign algorithm, which single-handedly reduced the time taken by SwapEdges twofold.

Table 4: Times for one swap per edge

Name	24 Threads	8 Threads	3 Threads	1 Thread	NetworkX
Kron	369.276 s	1136.710 s	2504.840 s	6724.940 s	-
Twitter	225.413 s	597.991 s	1052.390 s	4603.420 s	-
Road	4.030 s	9.685 s	15.710 s	41.154 s	1831.302 s
Urand	396.930 s	969.920 s	1877.130 s	5729.430 s	-
Web	326.222 s	679.517 s	1719.920 s	5518.480 s	-
Youtube	0.397 s	0.715 s	1.424 s	4.007 s	334.810 s
LiveJournal	5.232 s	13.065 s	20.996 s	53.820 s	2837.333 s
Orkut	16.008 s	31.106 s	60.518 s	155.844 s	15051.378 s

4. CONCLUSION

4.1 Code Simplicity in GraphBLAS

This project demonstrated the power of the SuiteSparse: GraphBLAS library to make simple yet effective code. The math behind the RCC algorithm is relatively simple: it essentially does two simple multiplications and then counts the values. This made the algorithm simple to write, easy to understand, and extremely fast because of the work done by SSGB under the hood. Most of the RCC code was done through simple calls to GraphBLAS, meaning that SSGB was able to take care of parallelism and other optimization work while the LAGraph code could focus on the higher-level aspects of the algorithm. The swap edges algorithm code was more complex, which allowed it to explore some of the more novel ways in which SSGB could be used, such as `LAGraph_FastAssign`. It even uncovered a couple of bugs in SSGB (which were all quickly fixed in the latest version of SSGB) and pointed to some algorithms that could be implemented into SSGB in the future. Finally, these algorithms will allow researchers to study much larger graphs than previously possible to uncover trends in many fields.

4.2 Future Work

The research into edge swap randomization has ignited many future ideas for other SSGB-based random graph generation algorithms, which could be useful for benchmarking new graph algorithms and forming different types of random graphs for network analysis. The FastAssign algorithm may inspire new SSGB methods for `GrB_Assign` and `GrB_Build`. The analysis of these new algorithms could greatly benefit from having another C implementation of both of them, so creating "naive" algorithms to compare these LAGraph algorithms to would be worthwhile. Adding more in-place matrix vector multiplication kernels to SSGB would greatly benefit FastAssign and make its performance much more robust. Specifically, an in-place kernel for bitmap outputs with accumulators would help avoid the workarounds needed in the SwapEdges code. The RCC algorithm would benefit from a parallel cumulative sum algorithm being added to SSGB because it currently uses a single-threaded cumulative sum. This cumulative sum could take a binary operator and apply it cumulatively on all the present entries in a vector or row in a matrix.

While `LAGraph_SwapEdges` is much faster than previous edge-swapping algorithms, it is expensive to swap 100 times per edge in a graph. One alternative is a matching algorithm mentioned by Milo and colleagues [5], which could now be more easily implemented into LAGraph. This algorithm starts with a random permutation to pair vertices into edges; this permutation could be achieved with a more faithful implementation of the algorithm presented by Green et. al [6]. The matching algorithm then uses an edge-swapping process similar to `LAGraph_SwapEdges` to fix parallel or self edges. This would use the process developed in this paper for finding parallel edges and would randomly swap them with other edges to attempt to remove them. Matching has the potential to be much faster than `LAGraph_SwapEdges` on larger graphs with the drawback that it may not be guaranteed to work on all graphs.

REFERENCES

[1] T. A. Davis, “Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra,” *ACM Trans. Math. Softw.*, vol. 45, Dec. 2019.

[2] J. Kepner, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, J. Moreira, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, and A. Lumsdaine, “Mathematical foundations of the graphblas,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, p. 1–9, IEEE, Sept. 2016.

[3] J. J. McAuley, L. da Fontoura Costa, and T. S. Caetano, “Rich-club phenomenon across complex network hierarchies,” *Applied Physics Letters*, vol. 91, Aug. 2007.

[4] V. Madhu, “Matching and coarsening in graphblas,” Master’s thesis, Texas A&M University, 2023.

[5] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon, “On the uniform generation of random graphs with prescribed degree sequences,” 2004.

[6] O. Green, C. Nolet, and J. Eaton, “Generating permutations using hash tables,” in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2022.

[7] S. Beamer, K. Asanović, and D. Patterson, “The gap benchmark suite,” 2017.

[8] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection.” http://snap.stanford.edu/data, June 2014.

APPENDIX A: RCC CODE

```
void iseq_2islt(int64_t *z, const int64_t *x, const int64_t *y)
{
    (*z) = (int64_t)((*x < *y) + (*x <= *y)) ;
}
void rich_club_formula(double *z, const int64_t *x, const int64_t *y)
{
    (*z) = ((double)(*x)) / (((double)(*y)) * (((double)(*y)) - 1.0));
}
int LAGraph_RichClubCoefficient
(
    // output:
    //rich_club_coefficents(i): rich club coefficent of i
    GrB_Vector *rich_club_coefficents,
    // input:
    LAGraph_Graph G, //input graph
    char *msg
)
{
    //--------------------------------------------------------------------------
    // Declorations
    //--------------------------------------------------------------------------
    . . .
    //--------------------------------------------------------------------------
    // Check inputs
    //--------------------------------------------------------------------------
    . . .
    //--------------------------------------------------------------------------
    // Initializations
    //--------------------------------------------------------------------------
    A = G->A ;
    GrB_BinaryOp_new(
        &iseq_2lt, &iseq_2islt, GrB_INT64, GrB_INT64, GrB_INT64) ;
    GrB_BinaryOp_new(
        &rcCalculation, (LAGraph_binary_function) (&rich_club_formula),
        GrB_FP64, GrB_INT64, GrB_INT64) ;
    GrB_Semiring_new(&plus_2le, GrB_PLUS_MONOID_INT64, iseq_2lt) ;

    //-------------------------------------------------------------------------
    // Calculations
    //-------------------------------------------------------------------------

    // degrees = G->out_degree - 1
    // Fill out degree vector, to target col_scale mxm on graphs
    // with singletons, scalar value irrelevant.
    GrB_Vector_assign_INT64(
        degrees, NULL, NULL, (int64_t) -1, GrB_ALL, 0, NULL) ;
    GrB_Vector_assign(
        degrees, NULL, GrB_PLUS_INT64, G->out_degree, GrB_ALL, 0, NULL) ;
```

```
GrB_Matrix_diag(&D, degrees, 0) ;

// Each edge in the graph gets the value of the degree of its row node
GrB_mxm (A_deg, NULL, NULL, GxB_ANY_FIRST_INT64, D, A, NULL) ;
// Sum the number of edges each node is "responsible" for.
GrB_mxv( node_edges, NULL, NULL, plus_2le, A_deg, degrees, NULL) ;


// The rest of this is indexing the number of edges and number of nodes
// at each degree and then doing a cummulative sum to know the amount
// of edges and nodes at degree geq k.
GrB_Vector_nvals (&edge_vec_nvals, node_edges) ;
GrB_Index vi_size = 0, vx_size = 0 ;
if(n == edge_vec_nvals)
{
    deg_x = degrees;
    degrees = NULL;
    node_edges_x = node_edges;
    node_edges = NULL;
}
else
{
    GrB_Vector_apply_BinaryOp2nd_INT64(
        degrees, NULL, NULL, GrB_MINUS_INT64, G->out_degree, 1, NULL
    ) ;
    GrB_Vector_new(&deg_x, GrB_BOOL, 0) ;
    GrB_Vector_new(&node_edges_x, GrB_BOOL, 0) ;
    GxB_Vector_extractTuples_Vector(NULL, deg_x, degrees, NULL) ;
    GxB_Vector_extractTuples_Vector(
        NULL, node_edges_x, node_edges, NULL
    ) ;
}
GrB_Vector_nvals(&edge_vec_nvals, node_edges_x) ;
GrB_Vector_new(&ones_v, GrB_INT64, edge_vec_nvals) ;
GrB_Vector_new(&ramp_v, GrB_INT64, edge_vec_nvals + 1) ;
GrB_Vector_assign_INT64(
    ramp_v, NULL, NULL, (int64_t) 0, GrB_ALL, 0, NULL) ;
GrB_apply (
    ramp_v, NULL, NULL, GrB_ROWINDEX_INT64, ramp_v, 0, NULL) ;
LAGraph_FastAssign (
    edges_per_deg, NULL, GrB_PLUS_INT64, deg_x, node_edges_x, ramp_v,
    GxB_PLUS_SECOND_INT64, NULL, msg
);
GrB_Vector_assign_INT64(
    ones_v, NULL, NULL, (int64_t) 1, GrB_ALL, 0, NULL) ;
LAGraph_FastAssign (
    verts_per_deg, NULL, NULL, deg_x, ones_v, ramp_v,
    GxB_PLUS_PAIR_INT64, NULL, msg
) ;

// Cumulative sum (TODO: should be a GBLAS method!)
GrB_Vector_nvals(&edge_vec_nvals, edges_per_deg) ;
GrB_Vector_extractTuples_INT64(
    vpd_index, deg_vertex_count, &edge_vec_nvals, verts_per_deg
    ) ;

    //run a cumulative sum (backward) on deg_vertex_count
    for(GrB_Index i = edge_vec_nvals - 1; i > 0; --i)
    {
        deg_vertex_count[i-1] += deg_vertex_count[i] ;
        edges_per_deg_arr[i-1] += edges_per_deg_arr[i] ;
    }
    GrB_Vector_clear(edges_per_deg) ;
    GrB_Vector_clear(verts_per_deg) ;
    GrB_Vector_build_INT64(
        edges_per_deg, epd_index, edges_per_deg_arr, edge_vec_nvals, NULL
    ) ;
    GrB_Vector_build_INT64(
        verts_per_deg, vpd_index, deg_vertex_count, edge_vec_nvals, NULL
    ) ;

    //Computes the RCC of a matrix
    GrB_eWiseMult(
        *rich_club_coefficents, NULL, NULL, rcCalculation,
        edges_per_deg, verts_per_deg, NULL
    ) ;

    return (GrB_SUCCESS) ;
}
```

APPENDIX B: FAST_ASSIGN CODE

```
int LAGraph_FastAssign_Monoid
(
    // outputs
    // Vector to be built (or assigned): initialized with correct dimensions.
    GrB_Vector c,
    // inputs
    const GrB_Vector mask,
    const GrB_BinaryOp accum,
    const GrB_Vector I_vec, // Indecies  (duplicates allowed)
    const GrB_Vector X_vec, // Values
    // Optional (Give me a ramp for faster calculations)
    const GrB_Vector ramp,
    const GrB_Monoid dup, // Applied to duplicates
    const GrB_Descriptor desc,
    char *msg
)
{
    GrB_BinaryOp op = NULL;
    GrB_Semiring sem = NULL;
    op = GrB_Second_[XTYPE] ;
    GrB_Semiring_new(&sem, dup, op) ;
    LAGraph_FastAssign_Semiring
        (c, mask, accum, I_vec, X_vec, ramp, sem, desc, msg) ;
    return (GrB_SUCCESS);
}

int LAGraph_FastAssign_Semiring
(
    // output
    // Vector to be built (or assigned): initialized with correct dimensions.
    GrB_Vector c,
    // inputs
    const GrB_Vector mask,
    const GrB_BinaryOp accum,
    const GrB_Vector I_vec, // Indecies  (duplicates allowed)
    const GrB_Vector X_vec, // Values
    // Optional (Give me a ramp for faster calculations)
    const GrB_Vector ramp,
    // monoid is applied to duplicates. Binary op should be SECOND.
    const GrB_Semiring dup,
    const GrB_Descriptor desc,
    char *msg
)
{
    //--------------------------------------------------------------------
    // Check inputs
    //--------------------------------------------------------------------
    LG_ASSERT (c != NULL, GrB_NULL_POINTER) ;
LG_ASSERT (I_vec != NULL, GrB_NULL_POINTER) ;
LG_ASSERT (X_vec != NULL, GrB_NULL_POINTER) ;
LG_ASSERT_MSG (c != X_vec, GrB_NOT_IMPLEMENTED,
    "c cannot be aliased with X_vec.") ;

//----------------------------------------------------------------------
// Find dimensions and type
//----------------------------------------------------------------------
GrB_Vector_size(&n, I_vec) ;
if(desc != NULL)
{
    GrB_get(desc, &trsp, GrB_INP0) ;
    if(trsp == GrB_TRAN)
    {
        GrB_Vector_size(&nrows, X_vec) ;
    }
    else
    {
        GrB_Vector_size(&nrows, c) ;
    }
}
else
{
    GrB_Vector_size(&nrows, c) ;
}
GrB_Vector_get_INT32(X_vec, (int32_t *) &iso, GxB_ISO) ;
GxB_Vector_type(&x_type, X_vec));

//----------------------------------------------------------------------
// Load up containers
//----------------------------------------------------------------------
GrB_Matrix_new(&P, GrB_BOOL, nrows, n));
GxB_Container_new(&con));
if(ramp == NULL)
{
    GrB_free(&(con->p)) ;
    ramp_type = (n + 1 <= INT32_MAX)? GrB_UINT32: GrB_UINT64;
    GrB_IndexUnaryOp idxnum = (n + 1 <= INT32_MAX)?
            GrB_ROWINDEX_INT32: GrB_ROWINDEX_INT64;
    GrB_Vector_new(&(con->p), ramp_type, n + 1));
    GrB_assign (con->p, NULL, NULL, 0, GrB_ALL, 0, NULL) ;
    GrB_apply (con->p, NULL, NULL, idxnum, con->p, 0, NULL) ;
}
else
{
    GxB_Vector_unload(
        ramp, &ramp_a, &ramp_type, &ramp_n, &ramp_size, &ramp_h, NULL) ;
    LG_ASSERT_MSG (ramp_n > n, GrB_DIMENSION_MISMATCH, "Ramp too small!");
    GxB_Vector_load(
        con->p, &ramp_a, ramp_type, n + 1, (n + 1) * (ramp_size / ramp_n),
        GxB_IS_READONLY, NULL) ;

    // Since con->p won’t free this array I should be safe to load it back
    // into ramp.
```

```
        GxB_Vector_load(
            ramp, &ramp_a, ramp_type, ramp_n, ramp_size, ramp_h, NULL) ;
        ramp_a = NULL;
    }
    if (c == I_vec)
    {
        GrB_Vector_dup(&con->i, I_vec) ;
    }
    else
    {
        // con->i = I_vec;
        GxB_Vector_unload(
            I_vec, &i_a, &i_type, &i_n, &i_size, &i_h, NULL) ;
        GxB_Vector_load(
            con->i, &i_a, i_type, i_n, i_size, GxB_IS_READONLY, NULL) ;
        // Since con->i won’t free this array I should be safe to load it back
        // into I_vec.
        GxB_Vector_load(
            I_vec, &i_a, i_type, i_n, i_size, i_h, NULL) ;
        i_a = NULL;
    }
    // con->x [0] = false, of length 1
    GrB_free(&(con->x)) ;
    GrB_Vector_new (&(con->x), GrB_BOOL, 1) ;
    GrB_assign (con->x, NULL, NULL, 0, GrB_ALL, 1, NULL) ;
    con->format = GxB_SPARSE;
    con->orientation = GrB_COLMAJOR;
    con->nrows = nrows;
    con->ncols = n ;
    con->nvals = n ;
    con->nrows_nonempty = -1 ;
    con->ncols_nonempty = n ;
    con->iso = true ;
    con->jumbled = false ;
    con->format = GxB_SPARSE ;
    con->orientation = GrB_COLMAJOR ;
    con->Y = NULL ;
    //----------------------------------------------------------------------
    // Load P and do the mxv
    //----------------------------------------------------------------------
    GxB_load_Matrix_from_Container(P, con, NULL));
    GrB_mxv(c, mask, accum, dup, P, X_vec, desc));
    //----------------------------------------------------------------------
    // Free work.
    // Note: this does not free inputs since they are marked GxB_IS_READONLY
    //----------------------------------------------------------------------
    GrB_free(&P) ;
    GrB_free(&con) ;
    return (GrB_SUCCESS) ;
}
```

APPENDIX C: SWAP_EDGES CODE

```
void shift_and
(uint16_t *z, const uint16_t *x)
{
    (*z) = (*x) & ((*x) << 8);
    (*z) |= (*z) >> 8;
}

typedef struct {
    uint64_t a;
    uint64_t b;
} edge_type64;

typedef struct {
    uint64_t a;
    uint64_t b;
    uint64_t c;
    uint64_t d;
} swap_type64;


typedef struct {
    uint32_t a;
    uint32_t b;
} edge_type32;


typedef struct {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
} swap_type32;

void swap_bc64
(swap_type64 *z, const swap_type64 *x, GrB_Index I, GrB_Index J, const bool *y)
{
    memcpy(z, x, sizeof(*z) ; //unnessesary when aliassed but done for safety.
    if(z->a == z->c || z->b == z->c || z->a == z->d || z->b == z->d ) return;
    if(I & 1)
    {
        uint64_t temp = z->d;
        z->d = z->b;
        z->b = temp;
    }
    else
    {
        uint64_t temp = z->c;
        z->c = z->b;
        z->b = temp;
    }
}
void swap_bc32
(swap_type32 *z, const swap_type32 *x, GrB_Index I, GrB_Index J, const bool *y)
{
    memcpy(z, x, sizeof(*z) ; //unnessesary when aliassed but done for safety.
    if(z->a == z->c || z->b == z->c || z->a == z->d || z->b == z->d ) return;
    if(I & 1)
    {
        uint32_t temp = z->d;
        z->d = z->b;
        z->b = temp;
    }
    else
    {
        uint32_t temp = z->c;
        z->c = z->b;
        z->b = temp;
    }
}

// using xorshift, from https://en.wikipedia.org/wiki/Xorshift
// with a state of uint64_t, or xorshift64star.
void hash_edge64
(uint64_t *z, const edge_type64 *x, const uint64_t *mask)
{
  (*z) = x->a ^ x->b;
 (*z) ^= (*z) << 13;
 (*z) ^= (*z) >> 7;
  (*z) ^= (x->a < x->b)? x->a: x->b;
 (*z) ^= (*z) << 17;
  (*z) &= (*mask);
}
void hash_edge32
(uint64_t *z, const edge_type32 *x, const uint64_t *mask)
{
  (*z) = x->a ^ x->b;
 (*z) ^= (*z) << 13;
 (*z) ^= (*z) >> 7;
  (*z) ^= (uint64_t)((x->a < x->b)? x->a: x->b);
 (*z) ^= (*z) << 17;
  (*z) &= (*mask);
}

void add_term
    (int8_t *z, const int8_t *x, const int8_t *y)
{
    (*z) = (*x) | (*y) + ((int8_t)1 & (*x) & (*y) ;
}


void edge2nd64
```

```
    (edge_type64 *z, const void *x, const edge_type64 *y)
{
    z->a = y->a;
    z->b = y->b;
}
void edge2nd32
    (edge_type32 *z, const void *x, const edge_type32 *y)
{
    z->a = y->a;
    z->b = y->b;
}


int LAGraph_SwapEdgesV2
(
    // output
    GrB_Matrix *A_new, //The adjacency matrix of G with edges randomly swapped
    // input: not modified
    LAGraph_Graph G,
    GrB_Index Q, // Swaps per edge
    char *msg
)
{
    //--------------------------------------------------------------------------
    // Declorations
    //--------------------------------------------------------------------------
    . . .
    //-------------------------------------------------------------------------
    // Check inputs
    //-------------------------------------------------------------------------
    LG_ASSERT_MSG (
        G->kind == LAGraph_ADJACENCY_UNDIRECTED,
        LAGRAPH_INVALID_GRAPH,
        "G must be undirected"
     ) ;
    // char type[LAGRAPH_MAX_NAME_LEN];
    LG_ASSERT_MSG (G->nself_edges == 0, LAGRAPH_NO_SELF_EDGES_ALLOWED,
        "G->nself_edges must be zero") ;
    LG_ASSERT (A_new != NULL, GrB_NULL_POINTER) ;

    //-------------------------------------------------------------------------
    // Initializations
    //-------------------------------------------------------------------------
    A = G->A ;
    // Types
    GrB_Matrix_nrows (&n, A) ;
    GrB_Matrix_new (&A_tril, GrB_BOOL, n, n) ;
    GrB_Vector_new(&Ai, GrB_BOOL, 0) ;
    GrB_Vector_new(&Aj, GrB_BOOL, 0) ;

    // Extract lower triangular edges.
    GrB_select (A_tril, NULL, NULL, GrB_TRIL, A, 0, NULL) ;
    GxB_Matrix_extractTuples_Vector(Ai, Aj, NULL, A_tril, NULL) ;
    GxB_Vector_type(&Ai_type, Ai));
```

```
int code;
GrB_get(Ai_type, &code, GrB_EL_TYPE_CODE);
GrB_Matrix_nvals(&e, A_tril) ;



//Init Operators -------------------------------------------------------------
if(code == GrB_UINT32_CODE)
{
    GxB_Type_new(
        &lg_edge, sizeof(edge_type32), "edge_type", EDGE_TYPE32) ;
    GxB_Type_new(
        &lg_swap, sizeof(swap_type32), "swap_type", SWAP_TYPE32) ;
    GxB_BinaryOp_new(
        &hash_seed_e, (GxB_binary_function) (&hash_edge32),
        GrB_UINT64, lg_edge, GrB_UINT64, "hash_edge", HASH_EDGE
    ) ;
    GxB_IndexUnaryOp_new (
        &swap_pair, (GxB_index_unary_function) (&swap_bc32),
        lg_swap, lg_swap, GrB_BOOL, "swap_bc", SWAP_BC32
    ) ;
    GxB_BinaryOp_new(
        &second_edge, (GxB_binary_function) (&edge2nd32),
        lg_edge, lg_edge, lg_edge, "edge2", EDGE2ND
    ) ;
    GxB_BinaryOp_new(
        &second_bool_edge, (GxB_binary_function) (&edge2nd32),
        lg_edge, GrB_BOOL, lg_edge, "edge2", EDGE2ND
    ) ;
}
else
{
    GxB_Type_new(
        &lg_edge, sizeof(edge_type64), "edge_type", EDGE_TYPE64) ;
    GxB_Type_new(
        &lg_swap, sizeof(swap_type64), "swap_type", SWAP_TYPE64) ;
    GxB_BinaryOp_new(
        &hash_seed_e, (GxB_binary_function) (&hash_edge64),
        GrB_UINT64, lg_edge, GrB_UINT64, "hash_edge", HASH_EDGE
    ) ;
    GxB_IndexUnaryOp_new (
        &swap_pair, (GxB_index_unary_function) (&swap_bc64),
        lg_swap, lg_swap, GrB_BOOL, "swap_bc", SWAP_BC64
    ) ;
    GxB_BinaryOp_new(
        &add_term_biop, (GxB_binary_function) (&add_term),
        GrB_INT8, GrB_INT8, GrB_INT8, "add_term", ADD_TERM
    ) ;
    GxB_BinaryOp_new(
        &second_edge, (GxB_binary_function) (&edge2nd64),
        lg_edge, lg_edge, lg_edge, "edge2", EDGE2ND
    ) ;
    GxB_BinaryOp_new(
        &second_bool_edge, (GxB_binary_function) (&edge2nd64),
        lg_edge, GrB_BOOL, lg_edge, "edge2", EDGE2ND
    ) ;
}

GxB_UnaryOp_new (
    &lg_shiftland, (GxB_unary_function) (&shift_and),
    GrB_UINT16, GrB_UINT16, "shift_and", SHIFT_AND
) ;
GxB_BinaryOp_new(
    &add_term_biop, (GxB_binary_function) (&add_term),
    GrB_INT8, GrB_INT8, GrB_INT8, "add_term", ADD_TERM
) ;

GxB_Monoid_terminal_new_INT8(
    &add_term_monoid, add_term_biop, (int8_t) 0, (int8_t) 2
) ;

edge_type64 iden_second = {0,0};
GrB_Monoid_new_UDT(
    &second_edge_monoid, second_edge, (void *) &iden_second
) ;

// Now working with the built-in ONEB binary op
GrB_Semiring_new(
    &plus_term_one, add_term_monoid, GrB_ONEB_INT8
) ;
GrB_Semiring_new(
    &second_second_edge, second_edge_monoid, second_bool_edge
) ;
// count swaps
GrB_Index num_swaps = 0, num_attempts = 0, swaps_per_loop = e / 3 ;

// Make E Matrix -------------------------------------------------------
GrB_Matrix_new(&E, Ai_type, e, 2) ;
GrB_Vector_new(&E_vec, Ai_type, 2 * e) ;

// Shuffle i and j into E_vec.
// Filling out E_vec helps assign be much quicker.
GrB_assign(
    E_vec, NULL, NULL, (int64_t) 0, GrB_ALL, 0, NULL));
GrB_Index stride[] = {(GrB_Index) 0, e * 2 - 1, (GrB_Index) 2} ;
GrB_Vector_assign(
    E_vec, NULL, NULL, Aj, stride, GxB_STRIDE, NULL) ;
stride[GxB_BEGIN] = 1;
GrB_Vector_assign(
    E_vec, NULL, NULL, Ai, stride, GxB_STRIDE, NULL) ;

GxB_Vector_unload(
    E_vec, &indices, &E_type, &e, &ind_size, &E_hand, NULL));
e /= 2;
GxB_Vector_load(
    E_vec, &indices, lg_edge, e, ind_size, E_hand, NULL));

// Find Hash Size -----------------------------------------------------
```

```
int shift_e ;
#if (!( defined ( __NVCC__) || defined ( __INTEL_CLANG_COMPILER) || \
    defined ( __INTEL_COMPILER) ) && defined ( _MSC_VER ))
// using the built-in Microsoft Windows compiler
// use ceil, log2, etc
shift_e = 63 - (int) floor (log2 ((double) e) ;
#else
shift_e = __builtin_clzl(e);
#endif
uint64_t ehash_size = (1ull << (67-shift_e) ;
printf("Hash Size: %ld", ehash_size);
GrB_Vector_new(&exists, GrB_INT8, ehash_size) ;
GrB_Vector_new(&hashed_edges, GrB_UINT64, e) ;

// Init Ramps -------------------------------------------------------------
GrB_Vector_new(&ramp_v, GrB_UINT64, e + 1) ;
GrB_Vector_assign_UINT64 (ramp_v, NULL, NULL, 0, GrB_ALL, 0, NULL) ;
GrB_Vector_apply_IndexOp_UINT64 (ramp_v, NULL, NULL,
    GrB_ROWINDEX_INT64, ramp_v, 0, NULL) ;
GrB_Index ramp_size;

// Init Constants ---------------------------------------------------------
GrB_Scalar_new (&one8, GrB_UINT8) ;
GrB_Scalar_setElement_UINT8 (one8, 1) ;
GrB_Vector_new (&x, GrB_BOOL, e) ;

// Make Random ------------------------------------------------------------
GrB_Vector_new(&random_v, GrB_UINT64, e) ;
GrB_Vector_new(&r_60, GrB_UINT64, e) ;
GrB_Vector_new(&r_permute, GrB_UINT64, 1ull << (64-shift_e)) ;
GrB_set (r_permute, GxB_BITMAP, GxB_SPARSITY_CONTROL) ;
GrB_set (exists, GxB_BITMAP | GxB_FULL, GxB_SPARSITY_CONTROL) ;
GrB_Vector_assign_UINT64 (
    random_v, NULL, NULL, 0, GrB_ALL, e, NULL) ;
LAGraph_Random_Seed(random_v, 1548945616ul, msg) ;

printf("Entering loop, Good Luck:\n") ;
while(num_swaps < e * Q && num_attempts < e * Q * 5)
{
    GrB_Index perm_size, arr_size, junk_size;
    // Coming into the loop:
    // E must be the incidence matrix of the new graph. W/o self edges nor
    // parallel edges. Each row must have exactly two distinct values.
    // random_v has a radom dense vector.
    GrB_Vector_apply_BinaryOp2nd_UINT64(
        r_60, NULL, NULL, GxB_BSHIFT_UINT64, random_v, -(shift_e), NULL
    ) ;
    GrB_Vector_clear(x) ;
    GrB_Vector_resize(x, e) ;
    GrB_Vector_assign_BOOL(
        x, NULL, NULL, true, GrB_ALL, 0, NULL) ;
    LG_TRY (LAGraph_FastAssign(
        r_permute, NULL, NULL, r_60, x, ramp_v, GxB_ANY_FIRSTJ_INT64,
        NULL, msg) ;
```

```
GrB_Index edges_permed = 0;
GrB_Vector_nvals(&edges_permed, r_permute) ;
GrB_Vector_new(&edge_perm, GrB_BOOL, edges_permed) ;
GrB_Vector_extractTuples(NULL, edge_perm, r_permute, NULL) ;
swaps_per_loop = LAGRAPH_MIN(swaps_per_loop, edges_permed / 2 ;

// Chose only the edges we need from  edge_perm
GrB_Vector_resize(edge_perm, swaps_per_loop * 2) ;

// Get the desired edges from the E_vec array.
GrB_Vector_new(&M, lg_edge, swaps_per_loop * 2) ;
GxB_Vector_extract_Vector(
    M, NULL, NULL, E_vec, edge_perm, NULL
) ;

// Make the swaps via the swap_pair unary op.
GxB_Vector_unload(
    M, (void **) &indices, &M_type, &M_nvals, &ind_size, &M_hand,
    NULL
) ;
GxB_Vector_load(
    M, (void **) &indices, lg_swap, M_nvals / 2, ind_size, M_hand, NULL
) ;
GrB_Vector_apply_IndexOp_BOOL(
    M, NULL, NULL, swap_pair, M, false, NULL) ;
GxB_Vector_unload(
    M, (void **) &indices, &M_type, &M_nvals, &ind_size, &M_hand,
    NULL
) ;
GxB_Vector_load(
    M, (void **) &indices, lg_edge, M_nvals * 2, ind_size, M_hand,
    NULL
) ;

// Hash Edges ---------------------------------------------------------
GrB_Vector_new(
    &new_hashed_edges, GrB_UINT64, swaps_per_loop * 2) ;

GrB_Vector_apply_BinaryOp2nd_UINT64(
    new_hashed_edges, NULL, NULL, hash_seed_e, M,
    ehash_size - 1ll, NULL
) ;
GrB_Vector_apply_BinaryOp2nd_UINT64(
    hashed_edges, NULL, NULL, hash_seed_e, E_vec,
    ehash_size - 1ll, NULL
) ;

//--------------------------------------------------------------------
// Build Hash Buckets
//--------------------------------------------------------------------

GrB_Vector_new(&dup_swaps_v, GrB_INT8, swaps_per_loop * 2) ;
GrB_set(dup_swaps_v, GxB_BITMAP, GxB_SPARSITY_CONTROL) ;
```

```
GrB_Vector_clear(x) ;
GrB_Vector_resize(x, e) ;
GrB_Vector_assign_BOOL(
    x, NULL, NULL, true, GrB_ALL, 0, NULL) ;
// place a one in any bucket that coresponds to an edge currently in E
LAGraph_FastAssign(
    exists, NULL, NULL, hashed_edges, x, ramp_v, GxB_ANY_PAIR_UINT8,
    NULL, msg
) ;

GrB_Vector_clear(x) ;
GrB_Vector_resize(x, swaps_per_loop * 2) ;
GrB_Vector_assign_BOOL(
    x, NULL, NULL, true, GrB_ALL, 0, NULL) ;

// exists cannot possibly be full at this point.
// Fill out exists in O(1) time.
GxB_Container_new(&con) ;
GxB_unload_Vector_into_Container(exists, con, NULL) ;
// Sanity check
LG_ASSERT (con->format == GxB_BITMAP, GrB_INVALID_VALUE ;
GrB_free(&exists);
exists = con->b;
con->b = NULL;
GrB_free(&con) ;
// exist has to be full at this point - confirmed by fprint
// GxB_print(exists, GxB_SUMMARY);

// "Count" all of the edges that fit into each bucket. Stop counting at
// 2 since we will have to throw that whole bucket away anyway.
LAGraph_FastAssign(
    exists, NULL, add_term_biop, new_hashed_edges, x, ramp_v,
    plus_term_one, NULL, msg
) ;
GrB_set (exists, GxB_BITMAP | GxB_FULL, GxB_SPARSITY_CONTROL) ;
// Select buckets with only one corresponding value
GrB_Vector_select_INT8(
    exists, NULL, NULL, GrB_VALUEEQ_UINT8, exists, (int8_t) 1,
    NULL
) ;

GxB_Vector_extract_Vector(
    dup_swaps_v, NULL, NULL, exists, new_hashed_edges, NULL) ;

// Fill out dup_swaps_v in O(1) time.
GxB_Container_new(&con) ;
GxB_unload_Vector_into_Container(dup_swaps_v, con, NULL) ;
n_keep = con->nvals;
GrB_free(&dup_swaps_v));
dup_swaps_v = con->b;
con->b = NULL;
GrB_free(&con) ;
GxB_Vector_unload(
        dup_swaps_v, (void **) &dup_swaps, &M_type, &M_nvals, &dup_arr_size,
        &M_hand, NULL) ;
    GxB_Vector_load(
        dup_swaps_v, (void **) &dup_swaps, GrB_INT16, M_nvals / 2,
        dup_arr_size, M_hand, NULL) ;
    GrB_apply(
        dup_swaps_v, NULL, NULL, lg_shiftland, dup_swaps_v, NULL) ;
    GxB_Vector_unload(
        dup_swaps_v, (void **) &dup_swaps, &M_type, &M_nvals, &dup_arr_size,
        &M_hand, NULL) ;
    GxB_Vector_load(
        dup_swaps_v, (void **) &dup_swaps, GrB_INT8, M_nvals * 2,
        dup_arr_size, M_hand, NULL) ;

    GrB_Vector_clear(exists) ;
    // -------------------------------------------------------------------
    // Place Good Swaps back into E_vec
    // -------------------------------------------------------------------
    GxB_Container_new(&con) ;
    GxB_unload_Vector_into_Container(M, con, NULL) ;
    GrB_free(&(con->b)) ;
    con->b = dup_swaps_v;
    con->nvals = n_keep;
    con->format = GxB_BITMAP;
    dup_swaps_v = NULL;
    GxB_load_Vector_from_Container(M, con, NULL) ;
    GrB_free(&con) ;
    LAGraph_FastAssign(
        E_vec, NULL, second_edge, edge_perm, M, ramp_v,
        second_second_edge, NULL, msg) ;


    n_keep /= 2;

    FREE_LOOP ; // Free Matricies that have to be rebuilt

    // Adjust number of swaps to do next.
    num_attempts += swaps_per_loop ;
    num_swaps += n_keep ;
    swaps_per_loop = n_keep * 3 ;
    swaps_per_loop = LAGRAPH_MAX(swaps_per_loop, 16) ;
    swaps_per_loop = LAGRAPH_MIN(swaps_per_loop, e / 3) ;

    LAGraph_Random_Next(random_v, msg) ;
    printf("#####Made %ld swaps. Total %ld out of %ld."\
         "Attempting %ld swaps next.#####\n\n",
         n_keep, num_swaps, e * Q, swaps_per_loop) ;
}
GxB_Vector_unload(
    E_vec, (void **) &indices, &lg_edge, &e, &ind_size, &E_hand, NULL)
GxB_Vector_load(
    E_vec, (void **) &indices, E_type, e * 2, ind_size, E_hand, NULL)
GrB_Vector_extract(
    Aj, NULL, NULL, E_vec, stride, GxB_STRIDE, NULL)
    stride[GxB_BEGIN] = 0;
    GrB_Vector_extract(
        Ai, NULL, NULL, E_vec, stride, GxB_STRIDE, NULL)
    // Build Output Matrix
    GrB_Matrix_new(A_new, GrB_BOOL, n, n) ;
    GxB_Matrix_build_Scalar_Vector(*A_new, Ai, Aj, one8, NULL) ;
    GrB_eWiseAdd(
        *A_new, NULL, NULL, GrB_LOR_MONOID_BOOL, *A_new,*A_new, GrB_DESC_T0
    ) ;

    LG_FREE_WORK ;
    return (GrB_SUCCESS) ;
}
```