EDGE BETWEENNESS CENTRALITY IN GRAPHBLAS

An Undergraduate Research Scholars Thesis

by

CASEY PEI

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 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, Casey Pei, 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

Page

ABSTRACT 1

DEDICATION 2

ACKNOWLEDGMENTS 3

NOMENCLATURE 4

1. INTRODUCTION 5

   1.1 Brandes' Traditional Algorithm 6
   1.2 The Exact Matrix EBC Algorithm 8

2. METHODS 11

   2.1 SuiteSparse:GraphBLAS 11
   2.2 General Implementation Approach 16
   2.3 Brandes' Traditional EBC Algorithm Implementation 16
   2.4 The Exact Matrix EBC Algorithm 19

3. RESULTS 25

   3.1 Testing Methods 25
   3.2 Accuracy 25
   3.3 Benchmarking 27

4. CONCLUSION 31

   4.1 Future Work 31

APPENDIX: BRANDE'S TRADITIONAL ALGORITHM 33

APPENDIX: EXACT GRAPHBLAS ALGORITHM 48

ABSTRACT

Edge Betweenness Centrality in GraphBLAS

Casey Pei
Department of Computer Science and Engineering
Texas A&M University

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

The Edge Betweenness Centrality (EBC) is a metric indicating that an edge can reach others on relatively short paths based on its ratio of total paths and shortest paths, showing the importance of the edge within a network. The EBC algorithm, proposed by Brandes in 2001, has wide-ranging applications in network analysis, community detection, and identifying key infrastructure in transportation and communication networks. While this method is widely used, it can be a bottleneck when applied to large-scale networks due to its high computational complexity. A more recent approach, developed by Robinson in 2011, adapts the EBC computation to leverage linear algebra techniques for improved performance, reducing the time complexity in certain cases. This paper presents an implementation of an exact matrix Edge Betweenness Centrality algorithm based on Robinson's approach using the SuiteSparse:GraphBLAS API in C, a powerful tool for performing matrix and vector operations on graphs. We demonstrate that while the linear algebra-based GraphBLAS implementation does not yet outperform Brandes' original algorithm for full EBC computation, it does show the utility of a linear algebra-based approach and reveals areas where the GraphBLAS kernels could be optimized.

DEDICATION

To my parents, Tiffany, and Ryan for supporting me.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Timothy Davis, for his invaluable guidance and support throughout the course of this research. In addition, I would like to thank my fellow undergraduate research cohort members Vidith Madhu, Daniel Pandyan, Darin Peries, Hemanth Mukesh, Gabriel Gomez, and Julio Dondisch for their solidarity and advice.

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 and books used for Edge Betweenness Centrality 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

Undergraduate research was supported by the Computer Science Department at Texas A&M University.

NOMENCLATURE

G, (V, E)	Graph
V, N	Number of nodes/vertices
E, M	Number of edges
ω	Weight function on the edges
$p(s, t)$	Path from s to t
$d(s, t)$	Distance between vertices s and t – i.e., the minimum length of any path connecting s and t in G
σ_{st}, σ_{ts}	Number of shortest paths from $s \in V$ to $t \in V$
$\sigma_{st}(v)$	Number of shortest paths from s to t that pass through some $v \in V$
$\sum_{s,t \in V} \frac{\sigma(s,t\|e)}{\sigma(s,t)}$	Betweenness centrality – a vertex can reach other vertices in relatively short paths, or that a vertex lies on considerable fractions of shortest paths connecting others
$\delta_{st}(v)$	Pairwise dependency – $\frac{\sigma_{st}(v)}{\sigma_{st}}$ of a pair $s, t \in V$ on an intermediary $v \in V$; the ratio of shortest paths between s and t that v lies on where: If v is on the shortest path between s, t (if $d_G(s,t) < d_G(s,v) + d_G(v,t)$): $\sigma_{st}(v)$ = 0 Else: $\sigma_{st}(v)$ = $\sigma_{sv} * \sigma_{vt}$

1. INTRODUCTION

In graphs, it is often useful to measure the importance of a vertex or edge compared to others for traversing the graph. This applies in social media, biology, material, scientific network analysis, and more. Several metrics have been designed, but one main metric is betweenness centrality, which was first proposed by Freeman in 1977 and Anthonisse in 1971 [1], [2]. In this paper, we specifically explore the edge betweenness centrality metric (EBC) and the algorithms used to achieve it.

$$EBC = \sum_{s,t \in V} \frac{\sigma(s,t|e)}{\sigma(s,t)} \tag{1}$$

Betweenness centrality of an edge is the a ratio of all paths that pass through it to the number of shortest paths that pass through it (see **Equation 1**). Therefore, an edge with a high betweenness centrality likely acts as a critical connection between two sections of the network, and removing it could disrupt communication between many pairs of nodes by severing their shortest paths. **Figure 1** provides an example with eight nodes in a network, in which a deeper red color represents a higher edge betweenness, and the edge with the highest edge betweenness centrality score is the connection between the connected subgraphs [3].

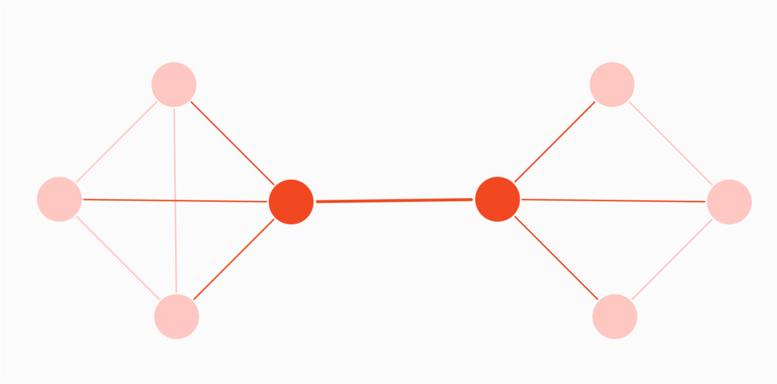


Figure 1: The edge betweenness centrality of an 8-node network

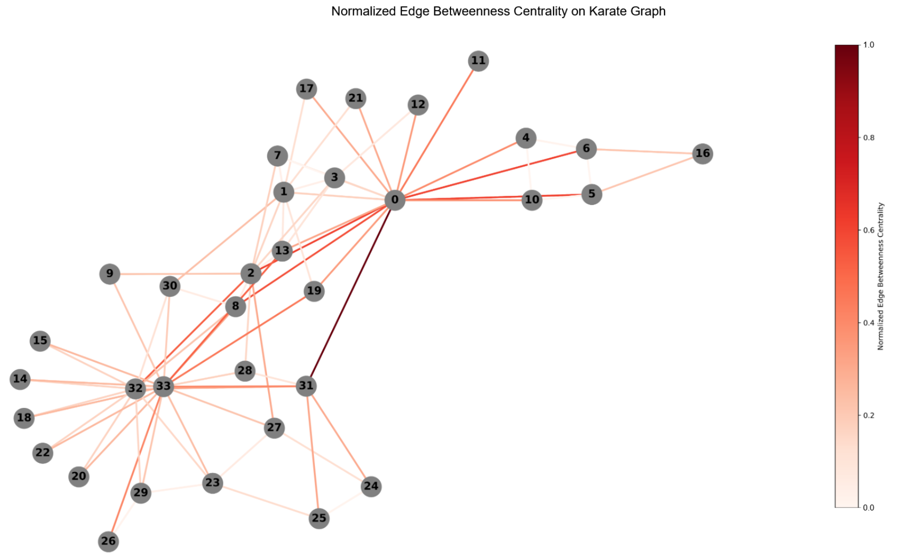

Normalized Edge Betweenness Centrality on Karate Graph
Normalized Edge Betweenness Centrality
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33

Figure 2: The edge betweenness centrality of the Karate graph

Figure 2 demonstrates the same edge betweenness centrality concept as **Figure 1** on the Karate Graph, again with the highest edge betweenness centrality score being the connection between the connected subgraphs. Here we see that the edge (0,31) has the highest edge betweenness metric.

However, often due to the size of these network graphs, it becomes prohibitively costly to use the EBC metric, as it grows more computationally expensive. Our goal is to implement an exact matrix EBC algorithm using the GraphBLAS framework, leveraging it and a linear algebra approach to improve computational cost. This implementation will be submitted into the LAGraph repository and benchmarked against an implementation of Brandes' algorithm in C as well as other libraries such as NetworkX.

1.1 Brandes' Traditional Algorithm

Brandes' algorithm became one of the most notable for edge betweenness centrality, after it was published in 2001. The algorithm can produce an exact result for the betweenness centrality of each edge in $O(NM)$ time complexity and in $O(M)$ space complexity [4]. The pseudocode of

Brandes' algorithm is shown in **Algorithm 1**.

Algorithm 1: Exact Traditional EBC in unweighted graphs

```
Result: Betweenness centrality $CB[v]$ for each vertex $v \in V$
for each $s \in V$ do
    $S \leftarrow$ empty stack;
    $P[w] \leftarrow$ empty list, $\forall w \in V$;
    $\sigma[t] \leftarrow 0, \ \forall t \in V$;
    $\sigma[s] \leftarrow 1$;
    $d[t] \leftarrow -1, \ \forall t \in V$;
    $d[s] \leftarrow 0$;
    $Q \leftarrow$ empty queue;
    enqueue $s$ into $Q$;
    while $Q$ is not empty do
        dequeue $v$ from $Q$;
        push $v$ into $S$;
        for each neighbor $w$ of $v$ do
            if $d[w] < 0$ then
                enqueue $w$ into $Q$;
                $d[w] \leftarrow d[v] + 1$;
            if $d[w] = d[v] + 1$ then
                $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
                append $v$ into $P[w]$;
    $\delta[v] \leftarrow 0, \ \forall v \in V$;
    while $S$ is not empty do
        pop $w$ from $S$;
        for each $v \in P[w]$ do
            $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v] \cdot (1 + \delta[w])}{\sigma[w]}$;
            if $w \neq s$ then
                $CB[w] \leftarrow CB[w] + \delta[w]$;
```

The algorithm is split into two major steps:

1. The first loop traverses the graph using breadth-first search (BFS) to determine the total number of shortest paths to each vertex.

2. The second loop backtracks in reverse depth order to perform centrality updates to each edge given the shortest paths found in the first step.

1.2 The Exact Matrix EBC Algorithm

Using linear algebra, we can parallelize the operations done in Brandes' algorithm, based on the algorithm presented by Robinson in 2011 [5] with some alterations to the calculation phase. This results in an exact matrix algorithm that has a time complexity of $O(N^2 + NM)$ and a space complexity of only $O(M)$. The pseudocode of this algorithm can be seen in **Algorithm 2**.

Algorithm 2: The Exact Matrix EBC Algorithm in unweighted graphs

```
Result: Resulting value for $B$
$B \leftarrow 0$;
for $r = 1$ to $N$ do
    $d \leftarrow 0$;
    $S \leftarrow 0$;
    $p \leftarrow 0, p(r) \leftarrow 1$;
    $U \leftarrow 0$;
    $v \leftarrow 0$;
    $f \leftarrow A(r, :)$;
    while $f \neq 0$ do
        $d \leftarrow d + 1$;
        $p \leftarrow p + f$;
        $S(d, :) \leftarrow f$;
        $f \leftarrow fA \times \neg p$;
    while $d \geq 1$ do
        $fd \leftarrow S(d, :)$;
        $fd1 \leftarrow S(d - 1, :)$;
        $J \leftarrow diag(fd, bc\_update, paths)$;
        $I \leftarrow diag(fd1 * p)$;
        $U \leftarrow I * A * J$;
        $B \leftarrow B + U$;
        $v \leftarrow U + .$;
        $d \leftarrow d - 1$;
```

The linear algebra approach uses the same method of having a BFS phase and a backtracking calculation phase. The improvements lie in utilizing matrix operations to perform the same operations in a more efficient manner.

1.2.1 The First Phase: BFS

To leverage breadth-first search through matrix-vector multiplication, it is essential to update the parent and path information for the entire depth in one search. Furthermore, to maintain the benefits of a linear algebra representation, the betweenness centrality updates should also be performed for the entire depth at once.

Instead of tracking the parents for the shortest paths, it is sufficient to record the depth of each vertex during the search. From this, the shortest path parents for a vertex v at breadth-first search depth d can be easily determined as $\forall u \in V$: depth$(u) = d - 1$ and $A(u, v) = 1$.

Line 9 performs the breadth-first search. After updating the search based on the new frontier obtained from the previous level, it selects the outgoing edges from that frontier, weights them by the number of shortest paths leading to their parents, and sums the values. It then filters out edges that lead to vertices already visited, resulting in the new frontier for the next depth level. The loop continues until no new vertices appear on the frontier.

1.2.2 The Second Phase: Calculating EBC

This phase differs from Robinson's approach because Robinson was using a different equation for EBC, whereas this has been edited to achieve the same result as Brandes' original algorithm [5].

In the second loop, the betweenness centrality updates involve dependencies only between parents and children, so updating an entire depth at once does not cause conflicts. Updates are carried out by selecting edges that come from vertices at the previous depth and point to vertices at the current depth. These edges, representing the betweenness centrality updates for their source vertices, are then weighted and summed accordingly.

Line 14 handles the betweenness centrality updates by processing the edges in reverse depth order. Initially, it computes the weights associated with the child vertices, filtering out edges that do not lead to vertices at the current depth. These weights are applied to the columns of the adjacency matrix. Next, the algorithm computes the weights related to the parent vertices, filtering out edges that do not originate from the previous depth, and applies these to the rows of the matrix. The updates are then added to the betweenness centrality scores, and the vertex flow is calculated by summing the rows of the current update.

2. METHODS

The traditional and GraphBLAS algorithms were entirely implemented in C, following the format of the other algorithms developed within the LAGraph library using its and GraphBLAS' methods. Preliminary comparisons of the algorithms to verify accuracy were made against NetworkX results in Python. The algorithms were then benchmarked using the Texas A&M Computer Science department's *BACKSLASH* system – which has Intel Xeon E5-2695 v2, 2.40GHz, 12 cores, one socket, 24 threads and a memory of 768 GB.

2.1 SuiteSparse:GraphBLAS

The GraphBLAS standard represents graph operations as operations on often sparse matrices and vectors based on semirings. This allows many graph algorithms to be completed in an inherently parallel fashion, such as breadth-first search [6]. As one can see comparing (**A**lgorithm 3) to (**A**lgorithm 4), GraphBLAS also uses masked assignment, which avoids if-statements in the innermost loops. This would otherwise require the algorithm to have access to the graph data structure at all times. Additionally, it allows GraphBLAS to avoid deeply nested loops such as the one in BFS in this case.

Algorithm 3: Traditional BFS

```
parent: vector of size n q: FIFO queue add s to q while q not empty do
    for each $i \in frontierq$ do
        for each edge (i,j) do
            if j not yet seen then
                add j to next q parent(j) = i flag j as seen
```

Accordingly, if an algorithm relies on BFS in some manner, there is clearly some potential speed-up that could be gained from using GraphBLAS. This is what originally higlighted EBC

Algorithm 4: GraphBLAS BFS

```
parent, q: vectors of size n
q(s) = 1
while q not empty do
    q( parent) = A' * q
    parent(q) = q
```

algorithms for implementation in GraphBLAS, since the first phase of these algorithms utilize BFS to get the shortest paths.

So by utilizing GraphBLAS, users can take advantage of parallel techniques common in linear algebra to enhance performance while reducing development time. The standard also offers other several benefits, including:

1. opaque data types; allowing users to focus on the algorithm rather than the implementation details,
2. enabling bulk operations on graphs without needing to manage individual nodes and edges,
3. portability; ensuring that if a faster implementation of GraphBLAS is released, the existing code will likely remain compatible without modification,

All this enables users to reduce development time while maintaining competitive performance.

2.1.1 Matrices and Vectors

One of the key concepts in understanding how GraphBLAS handles sparse matrices and vectors is its efficient storage and manipulation of data. Similar to the traditional representation of graphs using adjacency matrices, where the presence of an edge between nodes is represented by non-zero values, GraphBLAS leverages sparse matrices to store data in a way that avoids the inefficiency of dense storage. In GraphBLAS, a matrix or vector does not explicitly store zero values, which are often abundant in sparse data, such as graphs with many nodes but few edges.

In the case of sparse matrices, GraphBLAS uses opaque data structures to store only the non-zero entries, significantly reducing memory usage. Just like an adjacency matrix in graph theory, where a value at position $A(i, j)$ represents an edge from node i to node j, GraphBLAS stores values at positions that correspond to non-zero entries in the matrix. For example, in a sparse matrix, the presence of an entry at $A(i, j)$ indicates that some operation or relationship exists between the i-th and j-th elements. If an entry is zero, it is not stored, thus reducing the storage requirements and improving performance for large datasets.

When it comes to vectors, GraphBLAS follows a similar principle: only non-zero entries are stored. This is particularly useful for graphs with large numbers of nodes and sparse connectivity, where many nodes do not have edges. In this context, a vector's non-zero elements represent important relationships or data points, while the zero elements are implicitly excluded from the underlying data structure.

Furthermore, GraphBLAS does not require matrices or vectors to be square or full, as the underlying data structures are flexible enough to represent rectangular or sparse forms efficiently. For example, bipartite graphs, which involve two distinct sets of nodes, can be represented in GraphBLAS with non-square matrices, where only the relevant non-zero elements are stored.

The power of GraphBLAS lies in its abstraction from the underlying data structure. The user does not need to manage memory or storage concerns directly; instead, GraphBLAS handles the complexity of efficiently encoding sparse matrices and vectors using the best data structures for the task. This enables GraphBLAS to scale to large graphs and matrices, often with billions of non-zero entries, without the need for excessive memory usage.

2.1.2 Masks and Descriptors

Masks are opaque objects that influence the output of many operations. Masks can either be vectors or matrices, and they must match the dimensions of the output data structure of an operation. The primary purpose of a mask is to hide certain elements from appearing in the result. For instance, if a GraphBLAS operation attempts to assign a value to an index i, there must be a corresponding value at i in the mask. Regular masks require the value to be specifically `TRUE` for the mask to be applied correctly, whereas structural masks only need the presence of any value to activate. Structural masks are particularly useful as they ignore the actual values within the vector and only focus on the presence of values at specific locations. These masks are computationally more efficient than regular masks.

In addition to masks, input/output vectors and matrices in an operation can be modified using descriptors. Descriptors are lightweight flags that adjust how the input parameters are treated before the computation takes place. One of the most common descriptor operations is the complement, which reverses the effect of the mask. Instead of applying the mask, it will place values into the output array for any index not covered by the mask. Descriptors can also specify if a mask should be treated as a structural mask, whether an input vector or matrix should be transposed, or if the output should be entirely replaced by the result of the computation.

2.1.3 Semirings

A semiring consists of two main components: a monoid, which determines the operation used in place of traditional addition, and a multiplicative operator, which replaces the usual matrix multiplication.

In traditional matrix multiplication, the operation is based on the dot product of rows and columns, where elements in the rows of the first matrix are multiplied by corresponding elements in the columns of the second matrix and then summed. However, in GraphBLAS, the semiring replaces traditional matrix multiplication by substituting a new multiplicative operator and an additive monoid. For example, the plus-times semiring replaces traditional scalar multiplication with a custom binary operator and addition with a monoid, allowing for highly flexible operations. This allows operations such as subtraction, division, or user-defined operations instead of standard multiplication.

The ability to customize both the additive and multiplicative operators provides a higher level of abstraction, which is crucial for creating flexible and efficient graph algorithms. For example, the min-plus semiring replaces traditional addition with the minimum operation and multiplication with the plus operation, which is particularly useful for algorithms like shortest path.

Another important application of semirings is in matrix-vector multiplication, which can be used to find the neighbors of a node in a graph or to implement algorithms like BFS. These, and many other graph algorithms, can be built using either user-defined or built-in semirings.

2.1.4 Methods Used

The matrix implementation of the algorithm was done using specific functions from the GraphBLAS API specification. To give context for later explanations of this implementation, we must go through the functions used.

These functions provide a powerful set of operations for matrix and vector manipulation, specifically tailored for sparse graph algorithms. Each function is designed to perform operations efficiently on sparse matrices and vectors, which is important for large-scale graph computations.

1. **GrB_BinaryOp_new**: Creates a new binary operator in the GraphBLAS library.
2. **GrB_Matrix_nrows**: Returns the number of rows in a matrix.
3. **GrB_Vector_new**: Creates a new sparse vector of a specified type and size.
4. **GrB_Vector_nvals**: This function returns the number of non-zero values in a vector. It is used to check the size of the frontier during the BFS traversal in your code.
5. **GrB_Vector_clear**: Clears all entries in a vector. It is used to reset vectors during the BFS process.
6. **GrB_Col_extract**: Extracts a column from a matrix and stores it into a vector.
7. **GrB_vxm** (Matrix-Vector Multiply with Masking): Multiplies a vector with a matrix on a semiring and applies a optional mask.
8. **GrB_mxm** (Matrix-Matrix Multiply with Masking): Multiplies a matrix with a matrix on a semiring and applies an optional mask.
9. **GrB_eWiseMult** (Element-wise Multiply): Performs an element-wise multiplication between two matrices or vectors, applying a specified binary operator.

10. **GrB_Matrix_diag**: This function creates a diagonal matrix using the values from a vector.
11. **GrB_assign**: This function assigns values to elements of a matrix or vector based on a mask or indices.
12. **GrB_reduce**: This function reduces a matrix to a single vector by applying a monoid operation (like summing the values).
13. **GrB_eWiseAdd** (Element-wise Addition): Performs an element-wise addition between two matrices or vectors, applying a specified binary operator.

    GrB_Matrix_clear: Clears all values from a matrix, effectively setting it to zero.

2.2 General Implementation Approach

The general approach for all implementations is the same as the one demonstrated in the **Introduction** for **Algorithm 1**. That is the two main phases:

1. Phase 1: BFS to obtain shortest path counts
2. Phase 2: Back-tracking to calculate EBC for each edge

2.3 Brandes' Traditional EBC Algorithm Implementation

We implemented Brandes' traditional EBC algorithm in GraphBLAS as a way to verify accuracy of the matrix implementation. The implementation was relatively straightforward and similar to the pseudocode version presented in (**Algorithm 1**). To better describe the implementation details, we will only discuss what diverged compared to the pseudocode.

2.3.1 Data Structures

The algorithm utilizes the following data structures (given by the variable name used in the pseudocode and equations, and then by the name used in the code):

- $d[i]$ (called `depth[i]`): Distance array storing the shortest path depth from the source.

- $\sigma[i]$ (called `paths[i]`): Stores the count of shortest paths from the source to vertex i.
- S (called `S`): A stack used to facilitate dependency accumulation.
- $\delta[i]$ (called `bc_vertex_flow`): An array that accumulates the dependency scores.
- P (called `P`): A predecessor list tracking the shortest path tree.
- Q (called `queue`): A queue used for BFS traversal.

2.3.2 Phase 1: BFS to get shortest paths

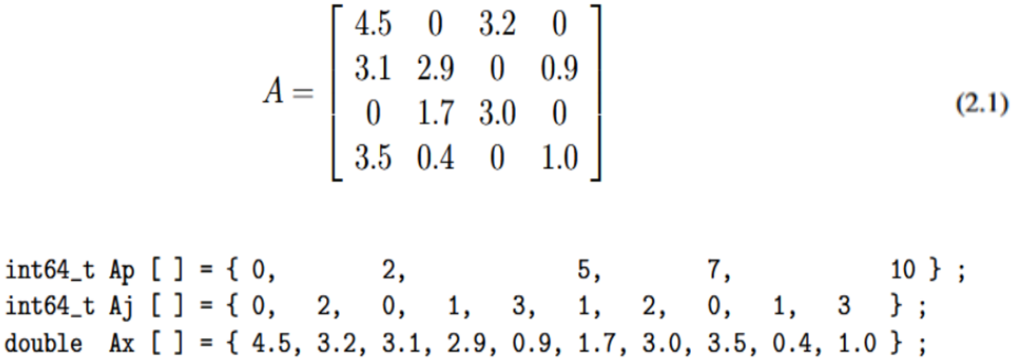
When implementing this algorithm efficiently in GraphBLAS, space optimization is crucial, especially for large sparse graphs. One effective space-saving measure we used was leveraging the *Compressed Sparse Row (CSR)* format to store the transposed adjacency matrix instead of allocating additional memory for tracking inbound edges separately.

2.3.2.1 Using CSR for the Transposed Adjacency Matrix

First, we needed to get the nodes of the predecessors' incoming edges rather than their outgoing edges for the backtracking phase to work. This is easily achieved by taking the transpose of the original adjacency matrix if the graph is directed (otherwise it is symmetric and there is no need).

CSR is a common storage format for sparse matrices, as it efficiently represents nonzero elements while reducing memory overhead. An adjacency matrix A (see **2.1**) in CSR format encodes outbound edges, where each row corresponds to a node, and its nonzero entries indicate outgoing edges to other nodes. An example of a matrix in CSR format can be found in **Figure 3**. The Ap array has size nrows+1, and determines the start and end of each row of the matrix. The ith row has entries in the columns given by the list Aj [Ap [i] ... Ap[i+1]-1], and the corresponding nonzero values are given by Ax [Ap [i] ... Ap[i+1]-1].

However, for certain graph algorithms such as BFS that require processing inbound edges, we need to work with the transpose A^T, where each row now represents incoming edges instead of outgoing ones.



$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix} \quad (2.1)$$

```
int64_t Ap [ ] = { 0,      2,            5,       7,           10 } ;
int64_t Aj [ ] = { 0,   2,   0,   1,   3,   1,   2,   0,   1,   3   } ;
double  Ax [ ] = { 4.5, 3.2, 3.1, 2.9, 0.9, 1.7, 3.0, 3.5, 0.4, 1.0 } ;
```

Figure 3: Diagram demonstrating CSR storage format of a matrix. Used with permission of Dr. Timothy Davis [7]

2.3.2.2 Reusing the Pointer to the p Array

By unpacking the A matrix in CSR format, it is easy to traverse the entries. For instance, it's able to travel through all the inbound edges of a node or all nodes of the A^T matrix via the A_p and A_j arrays respectively, since then the inbound edges are in $A_p[v]$. A significant benefit of this optimization was then avoiding the need for an additional data structure to store inbound edges, since we just pointed to the original entries of A.

Instead of creating a separate p array for A^T, the implementation simply reuses the p array of the original adjacency matrix A. Since transposing a sparse matrix only affects how edges are accessed but not their positions in memory, the original p array can still provide correct segmentations for traversing inbound edges efficiently.

2.3.3 Phase 2: Edge Betweenness Computation

The dependency score $\delta[v]$ is computed as follows:

$$\delta[v] = \sum_{w \in \mathrm{succ}(v)} \left(\frac{p[v]}{p[w]} (1 + \delta[w]) \right) \tag{2}$$

where $\mathrm{succ}(v)$ is the set of successor nodes of v.

Finally, the betweenness centrality score of an edge $(Update, v)$ is given by:

$$B[Update, v] = \sum_{r \in V} \delta[v] \tag{3}$$

2.4 The Exact Matrix EBC Algorithm

As stated previous this algorithm required more effort as the EBC calculation phase was novel to match the formula for EBC from Brandes. Therefore the calculation phase is also the most involved as BFS is straightforward in GraphBLAS.

2.4.1 Data Structures

The algorithm utilizes the following data structures (given by the variable name used in the pseudocode and equations, and then by the name used in the code):

- B (called `centrality`): The final matrix with the EBC value for each edge.
- S (called `Search`): Array of BFS search matrices. Search[i] is a sparse matrix that stores the depth at which each vertex is first seen thus far in each BFS at the current depth i. Each column corresponds to a BFS traversal starting from a source node.
- f (called `frontier`): Frontier vector, a sparse matrix. Stores the number of shortest paths to vertices at the current BFS depth.
- p (called `paths`): Paths matrix holds the number of shortest paths for each node and starting node discovered so far. A dense vector that is updated with sparse updates, and also used as a mask. Please note that p represents the same thing as σ as in the traditional algorithm pseudocode.
- v (called `bc_vertex_flow`): The betweenness centrality for each vertex. A dense vector that accumulates flow values during backtracking. Please note that v represents the same thing as δ as in the traditional algorithm pseudocode.
- U (called `Update`): Update matrix for betweenness centrality for each edge. A sparse matrix that holds intermediate centrality updates.

- `n/a` (called `Add_One_Divide`): Binary operator for computing $(1 + x)/y$ in centrality calculations.
- J (called `J_matrix`): Matrix for current level contributions.
- I (called `I_matrix`): Matrix for previous level contributions.

There are also some temporary variables that are only used in the code implementation and not the pseudocode:

- `Fd1A`: Intermediate product matrix.
- `temp_update`: Temporary vector for centrality updates.
- `J_vec`: Diagonal values for J_matrix.
- `I_vec`: Diagonal values for I_matrix.

2.4.2 Phase 1: BFS to get shortest path

Rather than having to unpack the AT matrix into CSR format, we can use matrix operations to traverse each column and obtain the paths and the frontier. This allows us to get each frontier as a group rather than iterate each node in the frontier sequentially, as well as accumulate the shortest path counts using a masked assignment with an addition semiring:

$$\sigma += frontier \tag{4}$$

```
GRB_TRY (GrB_assign (paths, NULL, GrB_PLUS_FP64, frontier,
    GrB_ALL, n, NULL)) ;
```

2.4.3 Phase 2: Edge Betweenness Computation

In the creation of this algorithm, we found that the EBC calculation presented by Robinson [5] used a different equation for the EBC metric. In order to match NetworkX and Brandes' conception of EBC [8], our implementation required a different calculation approach.

For the matrix implementation, we used matrix operations to achieve the same fundamental equation guiding this process:

$$\delta[v] \mathrel{+}= \sum_{w \in P[v]} \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w]) \tag{5}$$

where:

- δ represents the betweenness centrality update (bc_vertex_flow).
- $w = J$ corresponds to nodes in the current level.
- $v = I$ corresponds to the nodes at the previous level.
- σ represents the number of shortest paths ($paths$).

Each GraphBLAS operation in **Appendix: Exact Matrix Algorithm** helps construct the necessary matrices to efficiently implement the edge betweenness centrality update computation, **Equation 5**, using sparse matrix operations. The purpose and specific effect of each operation is summarized in **Table 1**.

This structured approach ensures efficient computation using sparse matrices while avoiding explicit loops, leveraging GraphBLAS' parallelized operations.

Table 1: GraphBLAS Operations and Their Mathematical Effects

Step	**GraphBLAS Operation**	**Mathematical Effect**
Compute J	GrB_eWiseMult + GrB_Matrix_diag	$J[w] = \frac{1+\delta[w]}{\sigma[w]}$
Compute I	GrB_Vector_extract + GrB_Matrix_diag	Isolates nodes at depth $d-1$
Compute $Fd1A$	GrB_mxm(Fd1A, I, A)	Temporary result of $I * A$
Compute U	GrB_mxm(Update, Fd1A, J)	Computes dependencies
Accumulate into B	GrB_eWiseAdd(B, U)	Updates BC scores
Sum U into Vector v	GrB_reduce(U), GrB_eWiseAdd(bc_vertex_flow)	Updates node BC flow

2.4.3.1 Constructing the J Matrix

The J matrix represents a diagonal matrix capturing the term:

$$J[w] = \frac{1+\delta[w]}{\sigma[w]} \tag{6}$$

The first operation computes element-wise division of bc_vertex_flow and $paths$, adding 1 to bc_vertex_flow. The result is then converted into a diagonal matrix to ensure correct application in the next step.

```
GRB_TRY (GrB_eWiseMult(J_vec, f_d, NULL, Add_One_Divide,
    bc_vertex_flow, paths, GrB_DESC_RS)) ;
GRB_TRY (GrB_Matrix_diag(&J_matrix, J_vec, 0)) ;
```

2.4.3.2 Constructing the I Matrix

The I matrix isolates nodes at the next depth level $d-1$, which ensures that the dependency values are correctly accumulated.

```
GRB_TRY (GrB_Vector_extract (I_vec, f_d1, NULL, paths,
    GrB_ALL, n, GrB_DESC_RS)) ;
```

```
GRB_TRY (GrB_Matrix_diag(&I_matrix, I_vec, 0)) ;
```

2.4.3.3 Computing $Fd1A = I \cdot A$

This operation computes inbound edges to nodes at depth $d-1$. It approximately serves as of a temporary matrix to obtain the left-hand side of the operation for calculating U.

```
GRB_TRY(GrB_mxm(Fd1A, NULL, NULL, LAGraph_plus_first_fp64,
    I_matrix, A, NULL)) ;
```

2.4.3.4 Computing $U = Fd1A \cdot J$

Compute the dependency accumulation step:

$$U[v] = \sum_{w \in P[v]} \sigma[v](\frac{1+\delta[w]}{\sigma[w]}) \tag{7}$$

```
GRB_TRY(GrB_mxm(Update, NULL, NULL,
    GrB_PLUS_TIMES_SEMIRING_FP64, Fd1A, J_matrix, NULL)) ;
```

2.4.3.5 Accumulating U into Betweenness Centrality

Update the betweenness centrality matrix:

$$B = B + U \tag{8}$$

```
GRB_TRY (GrB_assign(*centrality, A, GrB_PLUS_FP64, Update,
    GrB_ALL, n, GrB_ALL, n,
GrB_DESC_S)) ;
```

2.4.3.6 Summing U Into a Vector

Then we reduce U along columns to accumulate updates to v.

```
GRB_TRY (GrB_reduce(temp_update, NULL, NULL,
    GrB_PLUS_MONOID_FP64, Update, NULL)) ;
```

```
GRB_TRY (GrB_eWiseAdd(bc_vertex_flow, NULL, NULL,
    GrB_PLUS_FP64, bc_vertex_flow, temp_update, NULL)) ;
```

3. RESULTS

3.1 Testing Methods

As mentioned previously in the **Methods** section, the results were benchmarked on Texas A&M's *BACKSLASH* system, which has 12 cores, a 24 thread Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz, and 768GB of RAM. A typical maximum speedup on this platform is around the factor of 12, though our testing in this case did not use multiple threads so speedup was likely limited. The test suite consisted of 11 sparse graphs chosen from LAGraph/data:

1. *diamonds.mtx*,
2. *karate.mtx*,
3. *random_unweighted_general1.mtx*,
4. *random_unweighted_general2.mtx*,
5. *random_unweighted_bipartite1.mtx*,
6. *random_unweighted_bipartite2.mtx*,
7. *jagmesh7.mtx*,
8. *dnn_data/n1024-l1.mtx*,
9. *bcsstk13.mtx*,
10. *cryg2500.mtx*,
11. *pushpull.mtx*

All graphs were symmetric, had self-edges removed, were treated as unweighted even if they had weights.

3.2 Accuracy

The accuracy of the GraphBLAS exact matrix algorithm implementation was analyzed in two different methods. First, we verified using small graphs by comparing against the Python graph library, NetworkX (**Table 2**). This method of verification is limited, as NetworkX uses a traditional sequential algorithm implementation, which slows significantly on larger graphs and becomes impractical.

Table 2: Error on Brandes' Traditional EBC Algorithm in GraphBLAS vs NetworkX

Graph	**Error**
diamonds	0.0000E+00
karate	2.1312E-14

Secondarily, we also verified accuracy by comparing the results of the traditional algorithm to the exact matrix algorithm.

Table 3: Difference between Traditional and GraphBLAS algorithm

Graph	**Nodes**	**Edges**	**Difference**
diamonds	8	12	0.00E+00
karate	34	156	0.00E+00
random_unweighted_general1	50	208	1.42E-14
random_unweighted_general2	200	1912	2.84E-14
random_unweighted_bipartite1	300	2064	5.68E-14
random_unweighted_bipartite2	300	2056	5.68E-14
jagmesh7	1138	6312	2.91E-11
dnn_data/n1024-l1	1024	31744	0.00E+00
bcsstk13	2003	81880	7.28E-12
cryg2500	2500	9849	0.00E+00
pushpull	4000	194194	4.73E-10

If the total difference between all the EBC values of each edge between the two results was less than 1e-4, we found that the GraphBLAS algorithm was accurate. Observing **Table 3**, we see that for all the graphs tested, the algorithm was accurate. The differences in the results of the algorithms are merely due to differences in floating-point roundoff; all of these algorithms thus compute the same result.

3.3 Benchmarking

Before going into the benchmarks, we first compare the time and space complexity of the traditional and matrix implementation EBC algorithms to understand the general theoretical efficiency of these algorithms.

Table 4: Time and Space Complexity of Algorithms

Algorithm	**Time Complexity**	**Space Complexity**
Traditional (Brandes')	$O(NM)$	$O(M)$
Exact Matrix	$O(N^2 + NM)$	$O(M)$

Based on the general time complexities described in (**Table 4**), it appears that the matrix implementations of the EBC algorithm are less efficient. However, this is while not taking into account the efficiency added by the implementation details of SuiteSparse:GraphBLAS.

To measure efficiency changes resulting from GraphBLAS implementation, benchmark comparisons between Brandes' traditional algorithm and the exact matrix algorithm were run on Texas A&M University's **BACKSLASH** system (see **Table 5**).

Table 5: Performance of Traditional Brandes vs GraphBLAS Algorithm

			Time (s)	
Graph	**Nodes**	**Edges**	**Traditional**	**Exact GraphBLAS**
diamonds	8	12	1.8752E-03	7.6016E-03
karate	34	156	1.9718E-03	2.6832E-02
random_unweighted_general1	50	208	2.3977E-04	4.3072E-02
random_unweighted_general2	200	1912	4.6354E-03	1.7944E-01
random_unweighted_bipartite1	300	2064	1.0263E-02	3.8023E-01
random_unweighted_bipartite2	300	2056	1.0605E-02	3.5961E-01
jagmesh7	1138	6312	7.6609E-02	8.6047E+00
dnn_data/n1024-l1	1024	31744	2.7643E-01	1.0668E+01
bcsstk13	2003	81880	8.6999E-01	1.0994E+01
cryg2500	2500	9849	3.5900E-01	2.1005E+02
pushpull	4000	194194	5.9152E+00	7.0959E+02

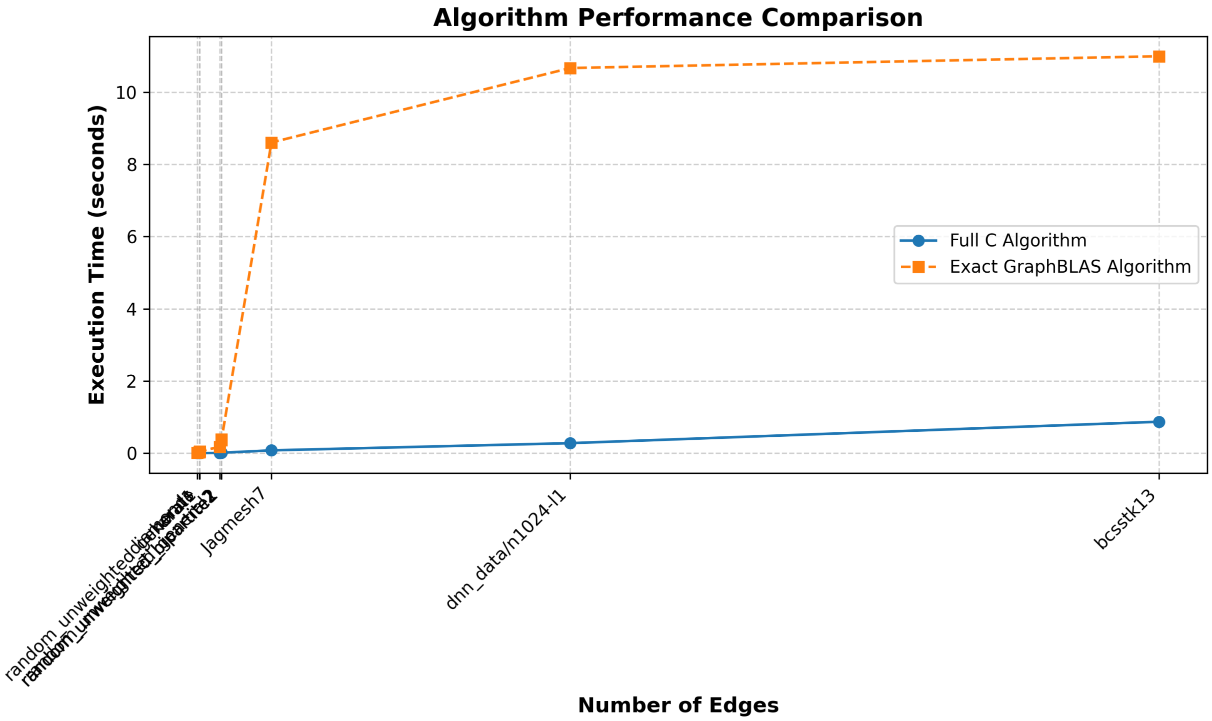


Figure 4: Performance of Traditional Brandes vs GraphBLAS Algorithm

As graph size expanded, we noted an unexpected increase in computation time (see **Figure 4**), which was significantly longer than initially anticipated. To identify the source of this issue, we utilized **Burble**, a profiling and performance analysis tool that will output a single line of output from each (significant) call to GraphBLAS. The Burble output is used to help detect when we are using sub-optimal methods.

Burble provided a detailed view of the computational performance and allowed us to identify that the problem seemed to arise mainly from the three following operations:

1. Matrix multiply (`GrB_mxm`) for the `I_matrix`.

```
GRB_TRY(GrB_mxm(Fd1A, NULL, NULL,
    LAGraph_plus_first_fp64, I_matrix, A,
    NULL)) ;
```

2. Matrix multiply (`GrB_mxm`) for the `J_matrix`.

```
GRB_TRY(GrB_mxm(Update, NULL, NULL,
    GrB_PLUS_TIMES_SEMIRING_FP64, Fd1A, J_matrix,
    NULL)) ;
```

3. Matrix Assignment (`GrB_assign`) when adding the `Update` matrix to the `centrality` matrix.

```
GRB_TRY (GrB_assign(*centrality, A, GrB_PLUS_FP64,
    Update, GrB_ALL, n, GrB_ALL, n,
    GrB_DESC_S)) ;
```

All of these operations were running slower than they should be expected to given how many elements they would actually have to work with due to their sparsity. For instance, we can look at the second operation with matrix multiplication involving the `J_matrix`.

Matrix multiplication involving a hypersparse matrix like `J_matrix` can be much faster, as the sparsity allows for skipping over zero elements, thus reducing unnecessary computations. Ideally, it would be. However, instead the kernel used for matrix multiplication in GraphBLAS was a general-purpose implementation.

This kernel processes every element of `Fd1A`, checking if each one would be multiplied with any non-zero element in `J_matrix`. Since `Fd1A` is not hypersparse, this results in performing redundant checks for many elements that do not interact with the hypersparsely populated `J_matrix`.

Burble's analysis revealed that the general-purpose kernel was inefficient when dealing with the sparsity of `J_matrix`. Instead of leveraging the sparsity of `J_matrix` to reduce the number of operations, the kernel was iterating over all the elements of the denser `Fd1A` matrix, leading to unnecessary calculations.

These inefficiencies became particularly evident even in smaller graphs. As the size of the graph increased, the computation time grew substantially, which significantly hindered performance. This can be seen in **Table 6** where as the graph size grows, the relative time these three operations take also grows, indicating that they are increasing asymptotically fast compared to the other operations used in the algorithm.

Table 6: Performance of GraphBLAS Operations in the Exact GraphBLAS algorithm

	% of total time taken			
Graph	**mxm with I**	**mxm with J**	**assign**	**total time (s)**
diamonds	4.39%	3.68%	7.70%	7.21E-03
karate	9.43%	8.77%	12.32%	2.24E-02
random_unweighted_general1	9.77%	8.91%	12.20%	3.74E-02
random_unweighted_general2	10.42%	15.46%	14.13%	1.50E-01
random_unweighted_bipartite1	10.10%	15.62%	15.35%	3.14E-01
random_unweighted_bipartite2	9.96%	15.53%	15.44%	3.02E-01
jagmesh7	10.54%	16.60%	12.01%	8.21E+00
dnn_data/n1024-l1	10.49%	20.45%	14.64%	1.05E+01
bcsstk13	10.22%	40.69%	17.24%	1.08E+01
cryg2500	10.56%	17.12%	11.51%	2.91E+01
pushpull	12.70%	10.93%	13.42%	7.06E+02

The performance issue observed in these operations is a key observation for future optimizations in GraphBLAS. Once optimized kernels for these special cases (such as in the case of hypersparse diagonal matrices when it comes to matrix multiplication, or assignment in place of a sparse matrix) are implemented, we expect significant improvements in computation time. These optimizations will make matrix-matrix multiplication and assignment much faster, especially for larger and sparser graphs.

4. CONCLUSION

This thesis overall demonstrates the utility of GraphBLAS in implementing a matrix version of a traditional algorithm, achieving results with less than 1E-10 difference compared to a traditional algorithm, which can be explained by rounding issues. We also found multiple performance bottlenecks in GraphBLAS when implementing the Exact EBC algorithm. They point to the need for at least three new specialized internal kernels in GraphBLAS, not a revision of the EBC algorithm or the GraphBLAS API. Once these new kernels are written, the performance of the EBC algorithm should be dramatically improved, as well as be able to be used by future algorithms implemented in GraphBLAS.

4.1 Future Work

As stated above, the performance bottlenecks identified in the matrix multiplication and assign operations provide valuable insights for future optimizations in the GraphBLAS library. By developing specialized kernels that take advantage of the sparsity inherent in matrices like `J_matrix`, we anticipate significant reductions in computation time. This optimization offers substantial performance gains, particularly as the size and sparsity of the graphs increase.

Additionally, there are more parameters that could be added to the EBC algorithms to increase functionality. Specifically, the ability to normalize the EBC values and work with weighted graphs, both of which are available in NetworkX. Adding normalization of the EBC values would be relatively simple, as it's just applying a scale to the EBC values after calculation, while increasing utility by having the values be from a 0 to 1 scale. As for running the EBC algorithm on weighted graphs, Robinson [5] notes that the traditional algorithm approach would use $O(N^2 log(N))$ time, with the matrix version having to take $O(N^3)$ time since the rows of the matrix cannot be stored and operated on as a priority queue. However, NetworkX implements phase 1 (BFS phase for unweighted graphs) using Dijkstra's algorithm, which uses a priority queue. Therefore, it may actually be possible to add this functionality with similar time complexity.

REFERENCES

[1] L. C. Freeman, “A set of measures of centrality based on betweenness,” *Sociometry*, vol. 40, p. 35, 1971.

[2] J. M. Anthonisse, “The rush in a directed graph. technical report,” *Stichting Mathematisch Centrum, Amsterdam*, 1971.

[3] L. Lu and M. Zhang, “Encyclopedia of systems biology,” in Springer, 2013, pp. 647–648. DOI: https://doi.org/10.1007/978-1-4419-9863-7_874.

[4] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[5] E. Robinson, “Graph algorithms in the language of linear algebra,” in J. Kepner and J. Gilbert, Eds. Society for Industrial and Applied Mathematics, 2011, ch. Chapter 6: Complex Graph Algorithms, pp. 68–84.

[6] T. Davis, *Juliacon 2023 keynote: Prof. tim davis*, [Online Video]. Available: https://youtu.be/0XZILz4hIkY?si=MJFxqWKdyiFr99H9, Cambridge, USA, Aug. 2023.

[7] T. Davis, *User guide for suitesparse:graphblas*, SuiteSparse:GraphBLAS, 2024, pp. 68–84.

[8] U. Brandes, “On variants of shortest-path betweenness centrality and their generic computation,” *Social Networks*, vol. 30, no. 2, pp. 136–145, 2008, ISSN: 0378-8733. DOI: https://doi.org/10.1016/j.socnet.2007.11.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378873307000731.

APPENDIX: BRANDE’S TRADITIONAL ALGORITHM

Listing A.1: LAGr_EdgeBetweennessCentrality: edge betweenness-centrality

```
//------------------------------------------------------------------------------
// LG_check_edgeBetweennessCentrality: reference implementation for edge
// betweenness centrality
//------------------------------------------------------------------------------

// LAGraph, (c) 2019-2022 by The LAGraph Contributors, All Rights Reserved.
// SPDX-License-Identifier: BSD-2-Clause
//
// For additional details (including references to third party source code and
// other files) see the LICENSE file or contact permission@sei.cmu.edu.  See
// Contributors.txt for a full list of contributors. Created, in part, with
// funding and support from the U.S. Government (see Acknowledgments.txt file).
// DM22-0790

// Contributed by Casey Pei, Texas A&M University

//------------------------------------------------------------------------------

#define LG_FREE_WORK                                \
{                                                   \
    LAGraph_Free ((void **) &queue, NULL) ;         \
    LAGraph_Free ((void **) &depth, NULL) ;             \
    LAGraph_Free ((void **) &bc_vertex_flow, NULL) ;         \
    LAGraph_Free ((void **) &S, NULL) ;             \
    LAGraph_Free ((void **) &paths, NULL) ;         \
    LAGraph_Free ((void **) &Pj, NULL) ;            \
    LAGraph_Free ((void **) &Ptail, NULL) ;         \
}

#define LG_FREE_ALL                                 \
{                                                   \
    LG_FREE_WORK ;                                  \
    LAGraph_Free ((void **) &Ap, NULL) ;            \
    LAGraph_Free ((void **) &Aj, NULL) ;            \
    LAGraph_Free ((void **) &Ax, NULL) ;            \
}

#include "LG_internal.h"
#include <LAGraphX.h>

//
```

```
------------------------------------------------------------------------------
// test the results from a Edge Betweenness Centrality
// ------------------------------------------------------------------------------

int LG_check_edgeBetweennessCentrality
(
    // output
    GrB_Matrix *C,      // centrality matrix
    // input
    LAGraph_Graph G,
    char *msg
)
{
    // --------------------------------------------------------------------------
    // initialize workspace variables
    // --------------------------------------------------------------------------

    double tt = LAGraph_WallClockTime ( ) ;
    GrB_Info info ;

    // Array storing shortest path distances from source to each vertex
    int64_t *depth = NULL ;

    // Array storing dependency scores during accumulation phase
    double *bc_vertex_flow = NULL ;

    // Stack used for backtracking phase in dependency accumulation
    int64_t *S = NULL ;

    // Queue used for BFS traversal
    int64_t *queue = NULL ;

    // Predecessor list components:
    // Pj: array of predecessor vertices
    // Ptail: end indices for each vertex's predecessor list
    // Phead: start indices for each vertex's predecessor list
    GrB_Index *Pj = NULL ;
    GrB_Index *Ptail = NULL ;
    GrB_Index *Phead = NULL ;

    // Array storing number of shortest paths to each vertex
    double *paths = NULL ;

    // Temporary array for centrality results
    double *result = NULL;

    // ---------------------------------------------------------------------------
    // check inputs
    //
```

```
   --------------------------------------------------------------------------

GrB_Index *Ap = NULL, *Aj = NULL, *neighbors = NULL, *ATp = NULL, *
   ATj = NULL ;
void *Ax = NULL, *ATx = NULL ;
GrB_Index Ap_size, Aj_size, Ax_size, n, nvals, ATp_size, ATj_size,
   ATx_size ;
LG_TRY (LAGraph_CheckGraph (G, msg)) ;
GRB_TRY (GrB_Matrix_nrows (&n, G->A)) ;
GRB_TRY (GrB_Matrix_nvals (&nvals, G->A)) ;
bool print_timings = (n >= 2000) ;

LG_TRY (LAGraph_DeleteSelfEdges (G, msg)) ;

GrB_Matrix A = G->A ;

LG_TRY (LAGraph_Cached_AT (G, msg)) ;

GrB_Matrix AT ;
if (G->kind == LAGraph_ADJACENCY_UNDIRECTED ||
     G->is_symmetric_structure == LAGraph_TRUE)
{
    // A and A' have the same structure
    // AT = A;
    GrB_Matrix_new (&AT, GrB_FP64, n, n) ;
    GrB_Matrix_dup (&AT, A) ;
}
else
```

```
{
    // A and A' differ
     AT = G->AT ;
     LG_ASSERT_MSG (AT != NULL, LAGRAPH_NOT_CACHED, "G->AT is required") ;
}


// ---------------------------------------------------------------------------

LG_CLEAR_MSG ;

// ---------------------------------------------------------------------------
// allocate workspace
// ---------------------------------------------------------------------------

LG_TRY(LAGraph_Malloc((void **)&depth, n, sizeof(int64_t), msg));

LG_TRY(LAGraph_Calloc((void **)&bc_vertex_flow, n, sizeof(double), msg));

LG_TRY(LAGraph_Malloc((void **)&S, n, sizeof(int64_t), msg));

LG_TRY(LAGraph_Malloc((void **)&queue, n, sizeof(int64_t), msg));

// ----------------------------------------------------------------------------
// bfs on the A
// ----------------------------------------------------------------------------

if (print_timings)
{
    tt = LAGraph_WallClockTime ( ) - tt ;
    printf ("LG_check_bfs init  time: %g sec\n", tt) ;
    tt = LAGraph_WallClockTime ( ) ;
}

// Initialize centrality matrix result to 0
// 1. result [(v, w)] <-- 0, for all (v, w) in E
// A temporary result centrality matrix initialized to 0 for all vertice,
// -- further changes would need to be made to make it a dictionary of edges.
GrB_Index result_size = n * n ;
LG_TRY(LAGraph_Calloc((void **)&result, result_size, sizeof(double), msg));

// result (v,w) is held in result (INDEX(v,w)):
#define INDEX(i,j) ((i)*n+(j))
```

```

// ----------------------------------------------------------------------------
// unpack the A matrix in CSR form for SuiteSparse:GraphBLAS
// ----------------------------------------------------------------------------

#if LAGRAPH_SUITESPARSE
bool iso, AT_iso ;
GRB_TRY (GxB_Matrix_unpack_CSR (A,
    &Ap, &Aj, &Ax, &Ap_size, &Aj_size, &Ax_size, &iso, NULL, NULL)) ;

GRB_TRY (GxB_Matrix_unpack_CSR (AT,
    &ATp, &ATj, &ATx, &ATp_size, &ATj_size, &ATx_size, &AT_iso, NULL, NULL)) ;
#endif

Phead = ATp ;

// ----------------------------------------------------------------------------

LG_TRY(LAGraph_Malloc((void **)&Pj, nvals, sizeof(GrB_Index), msg)) ;
LG_TRY(LAGraph_Malloc((void **)&Ptail, n, sizeof(GrB_Index), msg));
```

```
        // might need to be + 1

    LAGraph_Calloc ((void **) &paths, n, sizeof (double), msg) ;

    //
       =========================================================================

    // === Main computation loop
       ==============================================

    //
       =========================================================================


    // Process each vertex as a source
    for (int64_t s = 0; s < n; s++) {

        //
           ----------------------------------------------------------------------

        // Initialize data structures for current source
        //
           ----------------------------------------------------------------------


        size_t sp = 0;  // stack pointer
        memcpy(Ptail, ATp, n * sizeof(GrB_Index));

        // Initialize path counts
        for (int64_t i = 0; i < n; i++) {
```

```
        paths[i] = 0;
    }
    paths[s] = 1;

    // Initialize distances
    for (size_t t = 0; t < n; t++) {
        depth[t] = -1;
    }
    depth[s] = 0;

    // -----------------------------------------------------------------------
    // BFS phase to compute shortest paths
    // -----------------------------------------------------------------------

    int64_t qh = 0, qt = 0;  // queue head and tail
    queue[qt++] = s;         // enqueue source

    while (qh < qt) {
        int64_t v = queue[qh++];
        S[sp++] = v;

        // Process neighbors of current vertex
        for (int64_t p = Ap[v]; p < Ap[v+1]; p++) {
            int64_t w = Aj[p];

            // Handle unvisited vertices
            if (depth[w] < 0) {
                queue[qt++] = w;
                depth[w] = depth[v] + 1;
            }

            // Update path counts for vertices at next level
            if (depth[w] == depth[v] + 1) {
                paths[w] += paths[v];

                if (Ptail [w] >= Phead [w+1] || Ptail [w] < Phead [w])
                {
                    printf ("Ack! w=%ld Ptail [w]=%ld, Phead [w]=%ld Phead[w+1]=%ld\n",
                        w, Ptail [w], Phead [w], Phead [w+1]) ;
                    fflush (stdout) ; abort ( ) ;
                }

                Pj[Ptail[w]++] = v;
            }
        }
    }

    // ----------------------------------------------------------------------

    // Dependency accumulation phase
    //
```

```
           ----------------------------------------------------------------------

        // Initialize dependency scores
        for (size_t v = 0; v < n; v++) {
            bc_vertex_flow[v] = 0;
        }

        // Process vertices in reverse order of discovery
        while (sp > 0) {
            int64_t w = S[--sp];

            // Update dependencies through predecessors
            for (int64_t p = Phead[w]; p < Ptail[w]; p++) {
                int64_t v = Pj[p];

                // Compute and accumulate dependency
                double centrality = paths[v] * ((bc_vertex_flow[w] + 1)
                    / paths[w]);
                bc_vertex_flow[v] += centrality;
                result[INDEX(v,w)] += centrality;
            }
        }
    }

    if (print_timings)
    {
        tt = LAGraph_WallClockTime ( ) - tt ;
        printf ("LG_check_edgeBetweenessCentrality time: %g sec\n", tt)
```

```
            ;
        tt = LAGraph_WallClockTime ( ) ;
    }

    //--------------------------------------------------------------------------
    // repack the A matrix in CSR form for SuiteSparse:GraphBLAS
    //--------------------------------------------------------------------------

    #if LAGRAPH_SUITESPARSE
    GRB_TRY (GxB_Matrix_pack_CSR (A,
        &Ap, &Aj, &Ax, Ap_size, Aj_size, Ax_size, iso, false, NULL)) ;
    GRB_TRY (GxB_Matrix_pack_CSR (AT,
        &ATp, &ATj, &ATx, ATp_size, ATj_size, ATx_size, AT_iso, false, NULL)) ;
    #endif

#if 0
GrB_Info GxB_Matrix_pack_FullR  // pack a full matrix, held by row
(
    GrB_Matrix A,       // matrix to create (type, nrows, ncols unchanged)
    void **Ax,          // values, Ax_size >= nrows*ncols * (type size)
                        // or Ax_size >= (type size), if iso is true
    GrB_Index Ax_size,  // size of Ax in bytes
    bool iso,           // if true, A is iso
    const GrB_Descriptor desc
) ;
#endif

    GrB_Matrix C_temp;
    LG_TRY (GrB_Matrix_new(&C_temp, GrB_FP64, n, n)) ;
    LG_TRY (GxB_Matrix_pack_FullR(C_temp, (void **) &result,
       result_size * sizeof(double), false, NULL) ) ;

    LG_TRY (GrB_assign(C_temp, A, NULL, C_temp, GrB_ALL, n, GrB_ALL, n,
        GrB_DESC_RS)) ;

    *C = C_temp;

    //
       ---------------------------------------------------------------------------

    // free workspace and return result
    //
       ---------------------------------------------------------------------------


    LG_FREE_WORK ;

    if (print_timings)
    {
        tt = LAGraph_WallClockTime ( ) - tt ;
        printf ("LG_check_edgeBetweennessCentrality check time: %g sec\
           n", tt) ;
    }
    return (GrB_SUCCESS) ;
}
```

APPENDIX: EXACT GRAPHBLAS ALGORITHM

Listing B.1: LAGr_EdgeBetweennessCentrality: edge betweenness-centrality

```
// ------------------------------------------------------------------------------
// LAGr_EdgeBetweennessCentrality: edge betweenness-centrality
// ------------------------------------------------------------------------------

// LAGraph, (c) 2019-2022 by The LAGraph Contributors, All Rights Reserved.
// SPDX-Licene-Identifier: BSD-2-Clause
//
// For additional details (including references to third party source code and
// other files) see the LICEnE file or contact permission@sei.cmu.edu. See
// Contributors.txt for a full list of contributors. Created, in part, with
// funding and support from the U.S. Government (see Acknowledgments.txt file).
// DM22-0790

// Contributed by Casey Pei and Tim Davis, Texas A&M University;
// Adapted and revised from GraphBLAS C API Spec, Appendix B.4.

// ------------------------------------------------------------------------------

// LAGr_EdgeBetweennessCentrality: Exact algorithm for computing
// betweeness centrality.

// This is an Advanced algorithm (no self edges allowed)

// ------------------------------------------------------------------------------

#define useAssign
#define debug

#define LG_FREE_WORK                            \
{                                               \
    GrB_free (&frontier) ;                      \
    GrB_free (&J_vec) ;                         \
    GrB_free (&I_vec) ;                         \
    GrB_free (&J_matrix) ;                      \
    GrB_free (&I_matrix) ;                      \
    GrB_free (&Fd1A) ;                          \
    GrB_free (&paths) ;                         \
    GrB_free (&bc_vertex_flow) ;                     \
    GrB_free (&temp_update) ;                   \
    GrB_free (&Add_One_Divide) ;                \
    GrB_free (&Update) ;                     \
    if (Search != NULL)                              \
    {                                           \
        for (int64_t i = 0 ; i < n ; i++)       \
        {                                       \
            GrB_free (&(Search [i])) ;               \
        }                                       \
        LAGraph_Free ((void **) &Search, NULL) ;     \
    }                                           \
}

#define LG_FREE_ALL                 \
{                                   \
    LG_FREE_WORK ;                  \
    GrB_free (centrality) ;         \
}

#include "LG_internal.h"
#include <LAGraphX.h>

#undef  LAGRAPH_CATCH
#define LAGRAPH_CATCH(status)
                                             \
{

   \
    print ("LAGraph failure (file %s, line %d): status: %d",     \
        __FILE__, __LINE__, status) ;
                                             \
    LG_ERROR_MSG ("LAGraph failure (file %s, line %d): status: %d",
          \
        __FILE__, __LINE__, status) ;
                                                  \
    LG_FREE_ALL ;
                                                                     \
    return (status) ;
                                                                 \
}

#undef GRB_CATCH
#define GRB_CATCH(info)
                                                        \
{

   \
    printf ("GraphBLAS failure (file %s, line %d): info: %d",      \
        __FILE__, __LINE__, info) ;
                                                    \
    LG_ERROR_MSG ("GraphBLAS failure (file %s, line %d): info: %d",
          \
        __FILE__, __LINE__, info) ;
                                                    \
    LG_FREE_ALL ;
                                                                     \
    return (info) ;
                                                                   \
}

```

```
// --------------------------------------------------------------------------------
// (1+x)/y function for double: z = (1 + x) / y
// --------------------------------------------------------------------------------

void add_one_divide_function (double *z, const double *x, const double *y)
{
    double a = (*(x)) ;
    double b = (*(y)) ;
    (*(z)) = (1 + a) / b ;
}

#define ADD_ONE_DIVIDE_FUNCTION_DEFN                                        \
"void add_one_divide_function (double *z, const double *x, const double *y)\n" \
"{                                                                          \n" \
"    double a = (*(x)) ;                                                    \n" \
"    double b = (*(y)) ;                                                    \n" \
"    (*(z)) = (1 + a) / b ;                                                 \n" \
```

```c
"}"

// -----------------------------------------------------------------------------
// LAGr_EdgeBetweennessCentrality: edge betweenness-centrality
// -----------------------------------------------------------------------------

int LAGr_EdgeBetweennessCentrality
(
    // output:
    GrB_Matrix *centrality,     // centrality(i): betweeness centrality of i
    // input:
    LAGraph_Graph G,            // input graph
    char *msg
)
{

    // -------------------------------------------------------------------------
    // check inputs
    // -------------------------------------------------------------------------

    LG_CLEAR_MSG ;

    // Array of BFS search matrices.
    // Search[i] is a sparse matrix that stores the depth at which each vertex is
    // first seen thus far in each BFS at the current depth i. Each column
    // corresponds to a BFS traversal starting from a source node.
    GrB_Vector *Search = NULL ;

    // Frontier vector, a sparse matrix.
    // Stores # of shortest paths to vertices at current BFS depth
    GrB_Vector frontier = NULL ;

    // Paths matrix holds the number of shortest paths for each node and
    // starting node discovered so far. A dense vector that is updated with
    // sparse updates, and also used as a mask.
    GrB_Vector paths = NULL ;

    // The betweenness centrality for each vertex. A dense vector that
    // accumulates flow values during backtracking.
    GrB_Vector bc_vertex_flow = NULL ;

    // Update matrix for betweenness centrality for each edge. A sparse matrix
    // that holds intermediate centrality updates.
    GrB_Matrix Update = NULL ;
    // Binary operator for computing (1+x)/y in centrality calculations
    GrB_BinaryOp Add_One_Divide = NULL ;

    // Temporary vectors and matrices for intermediate calculations
    // Diagonal values for J_matrix
    GrB_Vector J_vec = NULL ;

    // Diagonal values for I_matrix
    GrB_Vector I_vec = NULL ;

    // Matrix for previous level contributions
    GrB_Matrix I_matrix = NULL ;

    // Matrix for current level contributions
    GrB_Matrix J_matrix = NULL ;

    // Intermediate product matrix
    GrB_Matrix Fd1A = NULL ;

    // Temporary vector for centrality updates
    GrB_Vector temp_update = NULL ;

    GrB_Index n = 0 ;                    // # nodes in the graph

    double t1_total = 0;
    double t2_total = 0;
    double t3_total = 0;

    LG_ASSERT (centrality != NULL, GrB_NULL_POINTER) ;
    (*centrality) = NULL ;
    LG_TRY (LAGraph_CheckGraph (G, msg)) ;

    GrB_Matrix A = G->A ;
    #if 0
    GrB_Matrix AT ;
    if (G->kind == LAGraph_ADJACENCY_UNDIRECTED ||
        G->is_symmetric_structure == LAGraph_TRUE)
    {
        // A and A' have the same structure
        AT = A ;
    }
    else
    {
        // A and A' differ
        AT = G->AT ;
        LG_ASSERT_MSG (AT != NULL, LAGRAPH_NOT_CACHED, "G->AT is required") ;
    }
    #endif

    // ==========================================================================
    // === initialization =====================================================
    // ==========================================================================
```

```c

    GRB_TRY (GxB_BinaryOp_new (&Add_One_Divide,
        (GxB_binary_function) add_one_divide_function,
        GrB_FP64, GrB_FP64, GrB_FP64,
        "add_one_divide_function", ADD_ONE_DIVIDE_FUNCTION_DEFN)) ;

    // Initialize the frontier, paths, Update, and bc_vertex_flow
    GRB_TRY (GrB_Matrix_nrows (&n, A)) ;
    GRB_TRY (GrB_Vector_new (&paths,    GrB_FP64, n)) ;
    GRB_TRY (GrB_Vector_new (&frontier, GrB_FP64, n)) ;
    GRB_TRY (GrB_Matrix_new (&Update, GrB_FP64, n, n)) ;
    GRB_TRY (GrB_Vector_new (&bc_vertex_flow, GrB_FP64, n)) ;


    // Initialize centrality matrix with zeros using A as structural mask
    LG_TRY (GrB_Matrix_new(centrality, GrB_FP64, n, n)) ;
    GRB_TRY (GrB_assign (*centrality, A, NULL, 0.0, GrB_ALL, n, GrB_ALL, n, GrB_DESC_S)) ;

    // Allocate memory for the array of S vectors
    LG_TRY (LAGraph_Calloc ((void **) &Search, n+1, sizeof (GrB_Vector), msg)) ;

    // =========================================================================

    // === Breadth-first search stage
```

```
==========================================
// =========================================================================

GrB_Index frontier_size, last_frontier_size = 0 ;
GRB_TRY (GrB_Vector_nvals (&frontier_size, frontier)) ;

int64_t depth, root ;
for (root = 0 ; root < n ; root++)
{
    depth = 0 ;

    // root frontier: Search [0](root) = true
    GrB_free (&(Search [0])) ;
    GRB_TRY (GrB_Vector_new(&(Search [0]), GrB_BOOL, n)) ;
    GRB_TRY (GrB_Vector_setElement_BOOL(Search [0], (bool) true, root)) ;

    // clear paths, and then set paths (root) = 1
    GRB_TRY (GrB_Vector_clear (paths)) ;
    GRB_TRY (GrB_Vector_setElement (paths, (double) 1.0, root)) ;

    GRB_TRY (GrB_Matrix_clear (Update)) ;

    // Extract row root from A into frontier vector: frontier = A(root,:)
    GRB_TRY (GrB_Col_extract (frontier, NULL, NULL, A, GrB_ALL, n, root,
        GrB_DESC_T0)) ;

    GRB_TRY (GrB_Vector_nvals (&frontier_size, frontier)) ;
    GRB_TRY (GrB_assign (frontier, frontier, NULL, 1.0, GrB_ALL, n,
        GrB_DESC_S)) ;

    while (frontier_size != 0)
    {
        depth++ ;

        //
           --------------------------------------------------------------------

        // paths += frontier
        // Accumulate path counts for vertices at current depth
        //
           --------------------------------------------------------------------


        GRB_TRY (GrB_assign (paths, NULL, GrB_PLUS_FP64, frontier,
           GrB_ALL, n,
            NULL)) ;

        //
           --------------------------------------------------------------------

        // Search[depth] = structure(frontier)
        // Record the frontier structure at current depth
        //
```

```
   ---------------------------------------------------------------------

GrB_free (&(Search [depth])) ;
LG_TRY (LAGraph_Vector_Structure (&(Search [depth]),
   frontier, msg)) ;

//
   ---------------------------------------------------------------------

// frontier<!paths> = frontier * A
//
   ---------------------------------------------------------------------

GRB_TRY (LG_SET_FORMAT_HINT (frontier, LG_SPARSE)) ;
GRB_TRY (GrB_vxm (frontier, paths, NULL, /*
   LAGraph_plus_first_fp64 */
    GxB_PLUS_FIRST_FP64, frontier,
    A, GrB_DESC_RSC )) ;

//
   ---------------------------------------------------------------------

// Get size of current frontier: frontier_size = nvals(
   frontier)
//
   ---------------------------------------------------------------------
```

```

    last_frontier_size = frontier_size ;
    GRB_TRY (GrB_Vector_nvals (&frontier_size, frontier)) ;
}


//
   =======================================================================
// === Betweenness centrality computation phase
   ============================
//
   =======================================================================

// bc_vertex_flow = ones (n, n) ; a full matrix (and stays full
   )
GRB_TRY (GrB_Vector_new (&bc_vertex_flow, GrB_FP64, n)) ;
GRB_TRY (GrB_assign(bc_vertex_flow, NULL, NULL, 0.0, GrB_ALL, n
   , NULL)) ;

GRB_TRY (GrB_Vector_new(&J_vec, GrB_FP64, n)) ;
GRB_TRY (GrB_Vector_new (&I_vec, GrB_FP64, n)) ;
GRB_TRY (GrB_Matrix_new (&Fd1A, GrB_FP64, n, n)) ;
GRB_TRY (GrB_Vector_new(&temp_update, GrB_FP64, n)) ; // Create
    a temporary vector

// Backtrack through the BFS and compute centrality updates for
    each vertex
// GrB_Index fd1_size;

// printf ("\n---------------------------- backtrack:\n") ;

while (depth >= 1)
{
    // printf ("\n---------------------------- backtrack depth : %" PRId64 "\n", depth) ;
    GrB_Vector f_d = Search [depth] ;
    GrB_Vector f_d1 = Search [depth - 1] ;

    // --------------------------------------------------------------------

    // j<S(depth, :)> = (1 + v) / p
    // J = diag(j)
    // Compute weighted contributions from current level
    // --------------------------------------------------------------------


    GRB_TRY (GrB_eWiseMult(J_vec, f_d, NULL, Add_One_Divide, bc_vertex_flow, paths, GrB_DESC_RS)) ;
    GRB_TRY (GrB_Matrix_diag(&J_matrix, J_vec, 0)) ;

    // --------------------------------------------------------------------

    // i<S(depth-1, :)> = p
```

```
// I = diag(i)
// Compute weighted contributions from previous level
//
   -------------------------------------------------------------------

GRB_TRY (GrB_Vector_extract (I_vec, f_d1, NULL, paths,
   GrB_ALL, n, GrB_DESC_RS)) ;
GRB_TRY (GrB_Matrix_diag(&I_matrix, I_vec, 0)) ;

//
   -------------------------------------------------------------------

// Update = I x A x J
// Compute edge updates based on current level weights
//
   -------------------------------------------------------------------

double t1 = LAGraph_WallClockTime();
GRB_TRY(GrB_mxm(Fd1A, NULL, NULL, LAGraph_plus_first_fp64,
    I_matrix, A, NULL));
t1 = LAGraph_WallClockTime() - t1;
t1_total += t1;

double t2 = LAGraph_WallClockTime();
GRB_TRY(GrB_mxm(Update, NULL, NULL,
   GrB_PLUS_TIMES_SEMIRING_FP64,
    Fd1A, J_matrix, NULL));
```

```
        t2 = LAGraph_WallClockTime() - t2;
        t2_total += t2;

        // -------------------------------------------------------------------
        // centrality<A> += Update
        // Accumulate centrality values for edges
        // -------------------------------------------------------------------

        #ifdef useAssign
            // centrality{A} += Update, using assign
            double t3 = LAGraph_WallClockTime();
            GRB_TRY (GrB_assign(*centrality, A, GrB_PLUS_FP64, Update, GrB_ALL, n, GrB_ALL, n,
            GrB_DESC_S));
            t3 = LAGraph_WallClockTime() - t3;
            t3_total += t3;
        #else
            // centrality = centrality + Update using eWiseAdd
            double t3 = LAGraph_WallClockTime();
            GRB_TRY (GrB_eWiseAdd (*centrality, NULL, NULL, GrB_PLUS_FP64, *centrality, Update, NULL));
            t3 = LAGraph_WallClockTime() - t3;
            t3_total += t3;
        #endif

        //
           ------------------------------------------------------------------

        // v = Update +.
        // Reduce update matrix to vector for next iteration
        //
           ------------------------------------------------------------------


        GRB_TRY (GrB_reduce(temp_update, NULL, NULL,
           GrB_PLUS_MONOID_FP64, Update, NULL)) ;
        GRB_TRY (GrB_eWiseAdd(bc_vertex_flow, NULL, NULL,
           GrB_PLUS_FP64, bc_vertex_flow, temp_update, NULL)) ;

        // 24 d = d - 1
        depth-- ;
    }


}

#ifdef debug
    printf("  I*A time: %g\n", t1_total);

    printf("  (I*A)*J time: %g\n", t2_total);

    #ifdef useAssign
        printf("  Centrality update using assign time: %g\n",
           t3_total);
        #else
            printf("  Centrality update using eWiseAdd time: %g\n", t3_total);
        #endif
    #endif


    // ==========================================================================

    // === finalize the centrality =============================================
    // ==========================================================================


    LG_FREE_WORK ;
    return (GrB_SUCCESS) ;
}
```