AN IMPLEMENTATION OF FAST GRAPHLET TRANSFORM IN GRAPHBLAS

An Undergraduate Research Scholars Thesis

by

TANNER HOKE

Submitted to the LAUNCH: 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 2022

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

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

TABLE OF CONTENTS

ABSTRACT

An Implementation of Fast Graphlet Transform in GraphBLAS

Tanner Hoke
Department of Computer Science and Engineering
Texas A&M University

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

Classically, many graph algorithms in computer science operate on representations of graphs other than the adjacency matrix. For example, the breadth-first search and many more complex shortest-path algorithms most often use an adjacency list representation of a graph. They also often involve iterating through some data structure which stores the current frontier of the search. However, many graph algorithms have dual implementations and perspectives involving linear algebraic operations on adjacency matrices. Approaching graph algorithms in this way comes with many advantages, including performance, ease of implementation, and code readability. GraphBLAS is a standard which provides a suite of building blocks for implementing graph algorithms using a linear algebraic approach via sparse matrices. Fast Graphlet Transform is an algorithm which detects graphlets, or subgraphs with small numbers of nodes, in a larger graph and describes the structure of the graph by determining the count of various types of graphlets adjacent to each vertex. In this paper, we provide some background on a linear algebraic perspective to graph algorithms and demonstrate its power by discussing the process of building the Fast Graphlet Transform algorithm for LAGraph, a library of graph algorithms built on top of GraphBLAS. Further, we compare both the ease of implementation and the performance of Fast Graphlet Transform in GraphBLAS with the code published by the original Fast Graphlet Transform paper authors [1].

ACKNOWLEDGMENTS

Contributors

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

The library code (GraphBLAS and LAGraph) used for An Implementation of Fast Graphlet Transform in GraphBLAS was provided by Dr. Timothy Davis.

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

Funding Sources

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

1. INTRODUCTION

1.1 Background

1.1.1 Graph Theory

A graph G is a mathematical structure consisting of a set $V(G)$ of objects, called vertices, and a set $E(G)$ of associations between pairs of vertices, called edges [2]. We typically write $G = (V, E)$. An edge $e \in E$ consists of a pair of vertices, typically called endpoints. In general, edges may be either directed or undirected, but for simplicity we consider only undirected graphs for now.

A subgraph of a graph G is a graph H whose vertices and edges are subsets of G's vertices and edges, respectively. Formally, we write $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ [2].

We may represent a graph pictorially by drawing its vertices as labelled points and its edges as lines connecting these points. As an example, let us denote by G the graph represented pictorially in Figure 1.1:

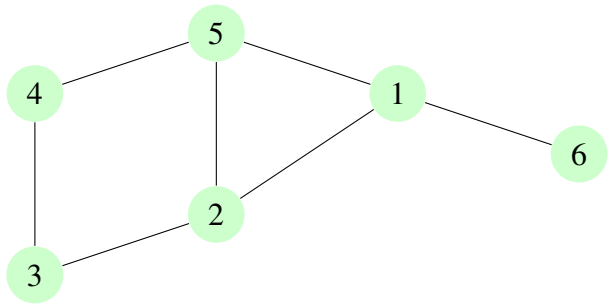

Figure 1.1: A drawing of the graph G.

As an example, we might also have a subgraph H of G represented pictorially:

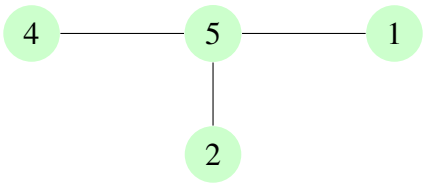

Figure 1.2: A drawing of the subgraph $H \subseteq G$.

Note that the relative positions of the vertices and edges in our graphical representation (Figure 1.2) do not matter; a graph is defined only in terms of its vertex set and edge set.

In computer science, we typically think of representing graphs using data structures in code. Common representations include the adjacency list, the edge list, and the adjacency matrix [3]. An adjacency list is a data structure that stores a list for each vertex in the graphs of its neighbors. Two vertices u and v are called neighbors if there is an edge whose endpoints are u and v. An edge list is a single list of all of the edges in the graph. For the graph G above, the list would be

$$\{(6,1),(4,5),(5,1),(1,2),(2,5),(2,3),(3,4)\}.$$

Finally, the adjacency matrix is a matrix $\mathbf{A}$ of size $|V| \times |V|$, where the entry $\mathbf{A}_{ij}$ is nonzero if and only if an edge exists between vertices i and j.[1] It is in this interpretation that we are most interested.

1.1.2 Linear Algebraic Perspective

At first glance, an adjacency matrix is not a particularly special representation of a graph. It has some algorithmic benefits; for example, one can decide if two vertices are neighbors in $O(1)$ (constant) time. It also has some drawbacks; for example, it takes $O(|V|)$ time to list the neighbors of any particular vertex. Our focus, though, is on the fact that it is a matrix, and by extension is amenable to linear algebraic methods.

Consider the simple problem of breadth-first search: starting from a vertex, visit all of the

[1] In a graph with edge weights, one typically stores the weight of the edge between vertices i and j.

vertices one edge away, then two edges away, and so on in the graph. Classically, this problem is solved in computer science using a loop and a queue data structure. We start by pushing the starting vertex onto the back of an empty queue. We subsequently repeat the following process while the queue is nonempty: visit a vertex at the front of the queue, iterate over its neighbors, and push those that we have not seen before onto the back of the queue [3].

For example, the following is a valid breadth-first search order starting from vertex 1 of the graph G (Figure 1.1) drawn above:

$$1, 6, 2, 5, 3, 4.$$

It turns out, though, that one can equivalently execute a breadth-first search by operating on the adjacency matrix with linear algebra. Let $\mathbf{A}$ be the adjacency matrix representation of a graph G, and u the vertex from which we begin the breadth-first search. Let $\mathbf{x}$ be a vector of dimension $|V|$, with the u-th entry equal to 1 and all others equal to 0. Then, if we compute

$$\mathbf{y} = \mathbf{A}^T\mathbf{x} \tag{Eq. 1.1}$$

using standard matrix-vector multiplication, $\mathbf{y}$ in Eq. 1.1 is a vector which has a 1 precisely at all vertices one step away from u in G. If we instead let

$$\mathbf{B} = \mathbf{A} + \mathbf{I} \tag{Eq. 1.2}$$

where $\mathbf{I}$ in Eq. 1.2 is the $|V| \times |V|$ identity matrix, then $\mathbf{y}' = (\mathbf{B})^T\mathbf{x}$ will consist of u and all its neighbors [4].

In general, computing

$$\mathbf{y} = (\mathbf{B}^T)^k\mathbf{x} \tag{Eq. 1.3}$$

results in a vector $\mathbf{y}$ consisting of all vertices in G that are at most distance k from u. Therefore, Eq. 1.3 represents precisely the steps of a breadth-first search.

Many well-known graph algorithms which are not classically implemented using adjacency

matrices are in fact approachable with linear algebra. Other examples include strongly connected components, shortest paths (Bellman-Ford, Floyd-Warshall), maximal independent set, graph contraction, graph partitioning, centrality metrics, minimum spanning tree, and more [4].

In many such cases, it is useful to extend the standard definition of matrix-vector multiplication to allow for various semirings. A semiring is a set of elements with two binary operations, typically called "addition" and "multiplication". They satisfy the following properties [4]:

- Addition and multiplication have identity elements, which we write as 0 and 1 respectively
- Both binary operations are associative
- Addition is commutative
- Multiplication distributes over addition from both sides
- The additive identity satisfies $0 * a = a * 0 = 0$

In the standard matrix-vector multiplication, we tend to use $\mathbb{R}$ with the classic addition and multiplication operations as our semiring. However, it is often useful to change the meaning of $+$ and $*$. For example, when implementing shortest path algorithms using linear algebra, the $\min . +$ semiring is the right choice [4]. Here, we replace the typical addition operation in matrix multiplication with the $\min$ function, and we replace the typical multiplication operation with the $+$ (standard addition) function.

1.1.3 GraphBLAS

GraphBLAS [5] is a standard which defines operations on sparse matrices and which is meant to be used to implement graph algorithms using linear algebraic methods. Many graphs are sparse, which informally means that most of the entries in their adjacency matrices are zero. Much work has been done on optimizing sparse matrix multiplication, as it is an important problem in a wide variety of fields.

GraphBLAS is intended to provide an interface for anyone to implement graph algorithms using linear algebra. SuiteSparse:GraphBLAS [6] is the reference C implementation, and is widely

used, including in MATLAB's default sparse matrix multiply. The implementation was created and is maintained by Dr. Timothy Davis at Texas A&M University.

Using this implementation, we will see that translating linear algebra written in math to code written in C is rather straightforward; we let GraphBLAS do most of the heavy lifting.

1.2 Motivation

1.2.1 Graphs

We have seen the mathematical definition of graphs, but it is also useful to note their generality and utility for many real-world problems. The birth of graph theory is often attributed to a problem posed by Leonhard Euler in 1736 called the Königsberg Bridge Problem [2]. The problem came from the real-world city of Königsberg, and dealt with the question of whether someone could leave the city, cross a series of seven bridges (which connected the city to an islands in the river along with the other city of the river) in some order, never use the same bridge twice, and end up back in the city.

Today, graphs are useful in solving an extraordinary number of problems. If we want to know the minimum cost to connect a network of cities with roads, the solution is a graph algorithm. If we want to know how to assign students to medical schools optimally, the solution is a graph algorithm. If we want some metric of "importance" for each computer in a network, we would want to use a graph algorithm, and so on.

Moreover, all of our vast social networks are themselves graphs of extraordinary size. In fact, many optimization problems that are faced in the real world involve very large graphs. However, as mentioned previously, it is often the case that we can take advantage of sparsity, and GraphBLAS is a state-of-the-art solution for handling such cases.

1.2.2 LAGraph

LAGraph [7] is a library of graph algorithms built on top of GraphBLAS to give users access to graph algorithms implemented via sparse matrices and linear algebra. The library exists as a reference and a resource for users, but also as a way of advancing the current state of graph

algorithms implemented using linear algebra.

LAGraph sits on top of GraphBLAS for ease-of-use. As an example, LAGraph defines a data structure called `LAGraph_Graph` which encapsulates the adjacency matrix for a graph as well as some of its other properties, such as whether the graph is directed or undirected. The data structure also deals with caching such properties as the transpose of the adjacency matrix and the row/column degrees so that these are not recomputed unnecessarily.

LAGraph is focused on providing users of graph algorithms with convenient implementations. At the same time, it does not sacrifice performance. When tested using the GAP benchmark suite [8], LAGraph's implementations are for most graphs extremely comparable and even in some cases faster. The difference is that while the GAP code is highly parallelized and difficult to read or write, LAGraph algorithms are short, readable, and easy to understand.

It is worth nothing that LAGraph is significantly slower than the GAP benchmark suite on one particular graph called the Road graph. The reason for this is discussed in the LAGraph introduction paper [7], but essentially it is because the Road graph has a very high diameter. This means that the largest distance between any two nodes is significantly high, and due to some internal memory overhead this slows down GraphBLAS' efficiency since every step of e.g. the BFS matrix multiply must do some extra work. This should be improved or even eliminated over time.

The LAGraph experimental branch is where new contributions to the library are created. Over time, LAGraph continues to amass implementations of important graph algorithms. Indeed, the contribution of this paper is an implementation of Fast Graphlet Transform [1] for LAGraph, and to this we now turn.

1.2.3 Fast Graphlet Transform

Fast Graphlet Transform [1] is a problem introduced by Floros, Pitsianis, and Sun which deals with counting the number of "graphlets" adjacent to every vertex in a graph. A graphlet in essence is a small subraph. In our running example (Figure 1.1), the triangle consisting of vertices 1,2, and 5 would be considered a graphlet. In the paper by Floros, Pitsianis, and Sun, a graphlet

dictionary consisting of sixteen graphlets Σ_{16} is introduced and discussed throughout. For our implementation in LAGraph, we use these same sixteen graphlets. Below are a few examples of graphlets which exist in Σ_{16} and appear in the graph G from Figure 1.1.

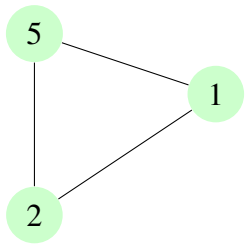

Figure 1.3: An occurrence of the graphlet σ_4 in G.

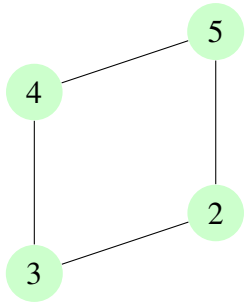

Figure 1.4: An occurrence of the graphlet σ_{12} in G.

The concept of graphlets was introduced by Pržulj, Corneil, and Jurisica in 2004 [9]. They discuss their use in detecting important properties of many real-world graphs, and in particular focus on biological applications. In fact, this first use of graphlet frequency analysis dealt with modeling protein-protein interaction in biology. They find that comparing graphlet frequencies in various models is valuable in testing which model is more accurate. In fact, they create a metric of similarity called the "relative graphlet frequency distance" which helps achieve this aim.

In general, counting the frequencies of incidence of a dictionary of graphlets for each vertex in a graph gives some idea of the structure of the graphs. For example, a common problem is

counting the number of triangles in a graph, of which Figure 1.3 is one example in G. Or maybe we want to know how many occurences of 4-cycles there are, and we would be interested in those subgraphs that look like Figure 1.4. Since Fast Graphlet Transform does this quickly and efficiently, one may think of it as a "summary" of a graph.

The Fast Graphlet Transform has been implemented in C++ and parallelized with OpenCilk by the authors of the original paper [10]. The implementation is efficient, but very difficult to code and requires familiarity with sparse matrix data structures and algorithms. However, the original paper gives simple linear algebraic formulations for each computation of graphlet frequency vectors. This implies that GraphBLAS may be suited to the task of easily writing code to implement Fast Graphlet Transform.

2. METHODS

In what follows, suppose we are working on an undirected, unweighted graph G with n vertices. Further, denote by A the $n \times n$ adjacency matrix of G.

In order to give a broad sense for the suitability of GraphBLAS to the problem of computing the Fast Graphlet Transform, we discuss in this section the implementation details for the computations of each graphlet frequency vector. In most cases, we will see that the translation of the linear algebraic formula to GraphBLAS code is simple and brief.

Note that the graphlet frequency vectors $\hat{d}_{0\ldots15}$ count the raw frequencies of each graphlet type incident to each vertex. Since some graphlets are subgraphs of other graphlets, this in some sense overcounts the frequencies of such graphlets. For example, consider that each triangle graphlet σ_4 consists of three separate bi-fork graphlets σ_3. It may be undesirable to count these three as occurrences of σ_3 since we are also counting them as one occurrence of σ_4. Thus, after computing the raw frequencies for each graphlet, we seek to convert the frequency matrix into one consisting of the net frequencies.

In order to compute net frequencies, we will use the approach mentioned in the original Fast Graphlet Transform paper [1]. Specifically, we take the inverse U_{16}^{-1} of the matrix U_{16} described in the paper. because the matrix is upper triangular with integer entries (the dictionary of graphlets is constructed in such a way that this holds true) its inverse will also have integer entries, so we do not need to worry about floating point error in converting raw graphlet frequencies to net graphlet frequencies. In fact, for the LAGraph implementation of the raw to net conversion, we simply pre-computed this matrix inverse and store it in code.

2.1 Computing $\hat{d}_0$

The vector $\hat{d}_0$ simply consists of n 1's. In other words, $\hat{d}_0 = e$. Implementing this in GraphBLAS is rather simple, as one would expect. The following code accomplishes the task:

```
GrB_Vector d_0 = NULL ;
```

```
GrB_TRY (GrB_Vector_new (&d_0, GrB_INT64, n)) ;
GrB_TRY (GrB_assign (d_0, NULL, NULL, 1, GrB_ALL, n, NULL)) ;
```

Here we introduce the `GrB_Vector` type, which is a column vector in GraphBLAS. We declare a vector of size n and fill it with 1 using `GrB_assign`.

2.2 Computing $\hat{d}_1$

The vector $\hat{d}_1$ consists of the degrees of vertices in G. In other words, the i-th entry of $\hat{d}_1$ gives the number of vertices adjacent to vertex i in G. Conveniently, LAGraph has a utility function which computes this vector, so we may simply call `LAGraph_Property_Row_Degree` in order to compute $\hat{d}_1 = Ae$:

```
GrB_Vector d_1 = NULL ;
GrB_TRY (LAGraph_Property_RowDegree (G, msg)) ;
d_1 = G->rowdegree ;
```

2.3 Computing $\hat{d}_2$

The vector $\hat{d}_2$ counts the number of 2-paths incident to each vertex. In particular, $\hat{d}_2 = p_2$, where $p_2 = Ap_1 - c_2$, $p_1 = \hat{d}_1$, and $c_2 = \hat{d}_1$ [1]. This first counts all paths of length two from a vertex, but this includes the cycles of length two (paths stepping from a vertex to its neighbor and back), so we then subtract the cycles of length two.

For our LAGraph implementation, since we have already computed $\hat{d}_1$, the operation is rather simple: a matrix multiplication and a vector subtraction. The following code implements this computation:

```
GrB_TRY (GrB_Vector_new (&d_2, GrB_INT64, n)) ;
GrB_TRY (GrB_mxv (d_2, NULL, NULL, GxB_PLUS_SECOND_INT64, A, d_1,
   NULL)) ;
GrB_TRY (GrB_eWiseMult (d_2, NULL, NULL, GrB_MINUS_INT64, d_2, d_1,
   NULL)) ;
```

Here, we perform the matrix-vector multiply $A\hat{d}_1$ using the `GrB_mxv` function. Next,

we use `GrB_eWiseMult` to subtract $\hat{d}_1$. The `GrB_eWiseMult` function operates on the set intersection of the patterns of the two vectors. Here, this is what we want, since $\hat{d}_1$ has the same sparsity pattern as $A\hat{d}_1$. In this case, we use `GrB_eWiseMult` and the `GrB_MINUS_INT64` monoid in order to perform vector-vector subtraction.

2.4 Computing $\hat{d}_3$

The vector $\hat{d}_3$ counts the frequency of incidence of the bi-fork graphlet for each vertex. We have $\hat{d}_3 = p_1 \odot (p_1 - 1)/2$. The $\odot$ operator is an element-wise multiplication. We could do this operation by first constructing the vector $p_1 - 1$, multiplying element-wise with p_1, then dividing by two. However, GraphBLAS also provides a mechanism for constructing user-defined unary operations. In this case, we can construct a unary operation f such that $f(x) = x(x-1)$, then apply that operation to the vector p_1. We will see that this unary operator will be helpful later too, so we will choose this approach.

First, we define the function `sub_one_mult` as follows:

```
void sub_one_mult (int64_t *z, const int64_t *x) { (*z) = (*x) *
   ((*x)-1) ; }
```

Now, we may use the function and `GrB_apply` to apply it to the vector $\hat{d}_1 = p_1$:

```
#define F_UNARY(f) ((void (*)(void *, const void *)) f)
GrB_UnaryOp Sub_one_mult = NULL ;
GrB_TRY (GrB_UnaryOp_new (&Sub_one_mult, F_UNARY (sub_one_mult),
   GrB_INT64, GrB_INT64)) ;

GrB_Vector d_3 = NULL ;
GrB_TRY (GrB_Vector_new (&d_3, GrB_INT64, n)) ;

GrB_TRY (GrB_apply (d_3, NULL, NULL, Sub_one_mult, d_1, NULL)) ;
GrB_TRY (GrB_apply (d_3, NULL, NULL, GrB_DIV_INT64, d_3, (int64_t) 2,
   NULL)) ;
```

Here, we use `F_UNARY` to cast the function to the type recognized by `GrB_UnaryOp_new`. Once we create the unary operator, we call `GrB_apply` using our new operator as discussed and also using the built in `GrB_DIV_INT64` operation to divide by two.

2.5 Computing $\hat{d}_4$

The vector $\hat{d}_4$ counts the frequency of incidence of triangles (cycles of length 3) for each vertex. We have $\hat{d}_4 = c_3 = C_3 e/2 = A \odot A^2 e/2$. Intuitively, we square A to get the cycles of length 2, and then we take the element-wise product with the matrix A itself to keep only those positions where we can "close" the path to form a triangle.

Rather than an element-wise product, though, one can interpret the operation as computing A^2, but using the structure of A as a mask for the matrix-matrix multiplication. The GraphBLAS matrix-matrix multiply function, `GrB_mxm`, takes an optional parameter to be used as the mask for the multiply, which speeds up such an operation significantly. So, rather than compute A^2 and then only keep the entries corresponding to the nonzero pattern of A, we only compute those entries of A^2 which match the pattern of A to begin with. The following code uses this approach:

```
GrB_Matrix C_3 = NULL ;
GrB_Vector d_4 = NULL ;

GrB_TRY (GrB_Matrix_new (&C_3, GrB_INT64, n, n)) ;
GrB_TRY (GrB_Vector_new (&d_4, GrB_INT64, n)) ;

// C_3 = hadamard(A, A^2)
GrB_TRY (GrB_mxm (C_3, A, NULL, GxB_PLUS_FIRST_INT64, A, A,
   GrB_DESC_ST1)) ;

// d_4 = c_3 = C_3e/2
GrB_TRY (GrB_reduce (d_4, NULL, NULL, GrB_PLUS_MONOID_INT64, C_3,
   NULL)) ;
GrB_TRY (GrB_apply (d_4, NULL, NULL, GrB_DIV_INT64, d_4, (int64_t) 2,
```

```
    NULL)) ;
```

Notice in particular the call to `GrB_mxm`. In previous calls to this function, we have passed `NULL` as the second parameter, which is the mask to apply to the computation. When `GrB_mxm` receives a `NULL` mask, it simply does standard matrix-matrix multiplication. However, since in this case we pass the matrix A itself as the mask, the function will only keep entries of A^2 where A itself has nonzero entries, as desired.

2.6 Computing $\hat{d}_5$

The vector $\hat{d}_5$ counts the frequency of incidence of vertices at an end of 3-paths. We have $\hat{d}_5 = p_3 = Ap_2 - p_1 \odot (p_1 - 1) - 2c_3$. Recall that we already have a `UnaryOp` to help with this Hadamard product, which we may reuse. For brevity, we now begin to omit code that initializes the relevant data structures:

```
// v = hadamard(p_1, p_1 - 1)
GrB_TRY (GrB_apply (v, NULL, NULL, Sub_one_mult, d_1, NULL)) ;

// two_c_3 = 2 * c_3 = 2 * d_4
GrB_TRY (GrB_apply (two_c_3, NULL, NULL, GrB_TIMES_INT64, 2, d_4,
    NULL)) ;

// d_5 = A * d_2
GrB_TRY (GrB_mxv (d_5, NULL, NULL, GxB_PLUS_SECOND_INT64, A, d_2,
    GrB_DESC_S)) ;

// d_5 -= hadamard(p_1, p_1 - 1)
GrB_TRY (GrB_eWiseAdd (d_5, NULL, NULL, GrB_MINUS_INT64, d_5, v,
    NULL)) ;

// d_5 -= two_c_3
```

```
GrB_TRY (GrB_eWiseAdd (d_5, NULL, NULL, GrB_MINUS_INT64, d_5, two_c_3,
   NULL)) ;
```

We will continue to use v as a vector to hold intermediate values in computations. In some cases, though, it makes sense to store temporary values in separate variables. Often, we will reuse these in later computations, so it makes sense to keep them around. For example, this is true of the vector `two_c_3`.

2.7 Computing $\hat{d}_6$

The vector $\hat{d}_6$ counts the frequency of incidence of vertices in an interior node of 3-paths. We have $\hat{d}_6 = p_2 \odot (p_1 - 1) - 2c_3$. We will keep $p_1 - 1$ as a vector itself, since it will be helpful for later computations.

```
// p_1_minus_one = p_1 - 1
GrB_TRY (GrB_apply (p_1_minus_one, NULL, NULL, GrB_MINUS_INT64, d_1,
   (int64_t) 1, NULL)) ;

// d_6 = hadamard(d_2, p_1-1)
GrB_TRY (GrB_eWiseMult (d_6, NULL, NULL, GrB_TIMES_INT64, d_2,
   p_1_minus_one, NULL)) ;

// d_6 -= 2c_3
GrB_TRY (GrB_eWiseAdd (d_6, NULL, NULL, GrB_MINUS_INT64, d_2, two_c_3,
   NULL)) ;
```

Note the use of `GrB_eWiseAdd` for the subtraction operation here and in the previous section. This is due to the fact that we can no longer guarantee that the only relevant subtractions occur at the intersection of the nonzero patterns of the two vectors. If we could, `GrB_eWiseMult` would actually be preferred, since it can take the sparse difference (only subtract two elements if both are nonzero). Recall that we can pass any monoid to `GrB_eWiseMult`.

2.8 Computing $\hat{d}_7$

The vector $\hat{d}_7$ counts the frequency of incidence of vertices at the leaf of a claw. We have $\hat{d}_7 = A((p_1 - 1) \odot (p_1 - 2))/2$.

```
GrB_TRY (GrB_apply (p_1_minus_two, NULL, NULL, GrB_MINUS_INT64, d_1,
   (int64_t) 2, NULL)) ;
GrB_TRY (GrB_eWiseMult (p_1_p_1_had, NULL, NULL, GrB_TIMES_INT64,
   p_1_minus_one, p_1_minus_two, NULL)) ;

GrB_TRY (GrB_mxv (d_7, NULL, NULL, GxB_PLUS_SECOND_INT64, A,
   p_1_p_1_had, NULL)) ;
GrB_TRY (GrB_apply (d_7, NULL, NULL, GrB_DIV_INT64, d_7, (int64_t) 2,
   NULL)) ;
```

2.9 Computing $\hat{d}_8$

The vector $\hat{d}_8$ counts the frequency of incidence of vertices at the root of a claw. We have $\hat{d}_8 = p_1 \odot (p_1 - 1) \odot (p_1 - 2)/6$. Note that we already computed $(p_1 - 1) \odot (p_1 - 2)$ in order to compute $\hat{d}_7$, and in GraphBLAS we may simply save this intermediate result in a `GrB_Vector` and reuse it to avoid recomputation.

```
GrB_TRY (GrB_eWiseMult (d_8, NULL, NULL, GrB_TIMES_INT64, d_1,
   p_1_p_1_had, NULL)) ;
GrB_TRY (GrB_apply (d_8, NULL, NULL, GrB_DIV_INT64, d_8, (int64_t) 6,
   NULL)) ;
```

2.10 Computing $\hat{d}_9$

The vector $\hat{d}_9$ counts the frequency of incidence of vertices at the handle tip of a paw. We have $\hat{d}_9 = Ac_3 - 2c_3$. Recall that $\hat{d}_4 = c_3$, so in our implementation we compute this as $A\hat{d}_4 - 2\hat{d}_4$.

```
GrB_TRY (GrB_mxv (d_9, NULL, NULL, GxB_PLUS_SECOND_INT64, A, d_4,
   NULL)) ;
```

```
GrB_TRY (GrB_eWiseAdd (d_9, NULL, NULL, GrB_MINUS_INT64, d_9, two_c_3,
   NULL)) ;
```

2.11 Computing $\hat{d}_{10}$

The vector $\hat{d}_{10}$ counts the frequency of incidence of vertices at a base node of a paw. We have $\hat{d}_{10} = C_3(p_1 - 2)$.

```
GrB_TRY (GrB_mxv (d_10, NULL, NULL, GxB_PLUS_TIMES_INT64, C_3,
   p_1_minus_two, NULL)) ;
```

2.12 Computing $\hat{d}_{11}$

The vector $\hat{d}_{11}$ counts the frequency of incidence of vertices at the center of a paw. We have $\hat{d}_{11} = (p_1 - 2) \odot c_3$.

```
GrB_TRY (GrB_eWiseMult (d_11, NULL, NULL, GrB_TIMES_INT64,
   p_1_minus_two, d_4, NULL)) ;
```

2.13 Computing $\hat{d}_{12}$

The vector $\hat{d}_{12}$ counts the frequency of incidence of 4-cycles at each vertex. We have $\hat{d}_{12} = c_4 = C_{4,2}e = P_2 \odot (P_2 - 1)e$, where $P_2 = A^2 - \text{diag}(d_1)$. This is the first case in which we must square the matrix A without any structural mask, making this the most costly computation discussed so far.

Computing the entire matrix A^2 is costly not only in terms of time, but also in terms of memory. Notice, though, that in fact we do not need the entire matrix at once. Instead, we seek in essence the sums across its columns (we end the computation by multiplying by the all 1's vector e). One way to take advantage of this is to compute only part of the matrix at a time. In fact, this is the approach taken in the original implementation of the Fast Graphlet Transform [10].

There is not a standard way to approach such a computation in GraphBLAS. One option is to go ahead and compute A^2 using GraphBLAS, and then call `GrB_reduce` to get it down to the vector form that we really want. This will work, but it will be rather slow, mostly due to the memory footprint.

The way we decide to solve this problem in our LAGraph implementation of Fast Graphlet Transform is to split the matrix into many smaller chunks by row, and then compute the answer for these rows in parallel. This is not a natural use of GraphBLAS, but it does work and is not particularly difficult to implement. For the sake of brevity, and because it is not GraphBLAS specific, we omit the code here.

2.14 Computing $\hat{d}_{13}$

The vector $\hat{d}_{13}$ counts the frequency of incidence of vertices at an off-cord of a diamond. We have $\hat{d}_{13} = D_{4,c}e/2 = A \odot (A(C_3 - A))$.

```
GRB_TRY (GrB_eWiseMult (D_4c, NULL, NULL, GrB_MINUS_INT64, C_3, A,
   NULL)) ;
GRB_TRY (GrB_mxm (D_4c, A, NULL, GxB_PLUS_SECOND_INT64, A, D_4c,
   GrB_DESC_S)) ;

// d_13 = D_{4,c}*e/2
GRB_TRY (GrB_reduce (d_13, NULL, NULL, GrB_PLUS_INT64, D_4c, NULL)) ;
GRB_TRY (GrB_apply (d_13, NULL, NULL, GrB_DIV_INT64, d_13, (int64_t)
   2, NULL)) ;
```

2.15 Computing $\hat{d}_{14}$

We take advantage of the fact that C_3, which we computed earlier, is the masked version of the matrix P_2, so that we may avoid doing the same work again. It is okay to use the masked version of P_2, since in the end we take the mask with A to compute $\hat{d}_{14} = D_{4,3}e/2 = A \odot C_{4,2} = A \odot P_2 \odot (P_2 - 1)$.

```
// P_2 = A*A - diag(d_1)
GRB_TRY (GrB_eWiseAdd (P_2, A, NULL, GrB_MINUS_INT64, C_3, D_1, NULL))
   ;

// C_42 = hadamard(P_2, P_2 - 1)
```

```
GRB_TRY (GrB_apply (C_42, A, NULL, Sub_one_mult, P_2, NULL)) ;

GRB_TRY (GrB_eWiseMult (D_43, NULL, NULL, GrB_TIMES_INT64, A, C_42,
   NULL)) ;

// d_14 = D_{4,3}*e/2
GRB_TRY (GrB_reduce (d_14, NULL, NULL, GrB_PLUS_INT64, D_43, NULL)) ;
GRB_TRY (GrB_apply (d_14, NULL, NULL, GrB_DIV_INT64, d_14, (int64_t)
   2, NULL)) ;
```

2.16 Computing $\hat{d}_{15}$

The vector $\hat{d}_{15}$ counts the frequency of incidence of vertices at a 4-clique, or in other words part of a subgraph isomorphic to K_4. This graphlet in particular seems to not have a very natural linear algebraic formulation, and as such is not well-suited to GraphBLAS. The equation given in the Fast Graphlet Transform paper is much more iterative, and in fact their implementation involves iterating over edges and common neighbors in the graph. For our LAGraph implementation, we chose to reproduce this approach using `GxB_Iterator`, but certainly this is not a GraphBLAS approach to the problem, and we omit the code here for brevity.

3. RESULTS

In order to test our implementation of Fast Graphlet Transfrom in GraphBLAS, we will benchmark our code on a variety of graphs against the implementation of Fast Graphlet Transform by the original paper authors [10].

For the sake of this paper, we will benchmark against two real-world graphs from the SNAP matrix collection [11]. The data was obtained from the SuiteSparse matrix collection [12].

Table 3.1: Benchmarks for our implementation of the Fast Graphlet Transform.

	com-Youtube	com-LiveJournal
LAGraph	19.523 sec	180.347 sec
fglt	7.4204 sec	34.518 sec
LAGraph (no d_{15})	15.830 sec	66.879 sec
fglt (no d_{15})	6.903 sec	29.855 sec
LAGraph (no d_{12}, d_{15})	2.455 sec	17.295 sec

As discussed previously, GraphBLAS is well-suited to all graphlet computations except those for d_{12} and d_{15}. We therefore benchmark the computation of all graphlets in Table 3.1, but also include times for our LAGraph implementation without d_{15}, and also without either d_{15} or d_{12}. Unfortunately, due to the fact that fglt computes d_{12} alongside many of the other frequency vectors, it would be a nontrivial modification to benchmark against fglt without d_{12}.

All benchmarks were conducted on a machine with an Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz.

3.1 The com-Youtube Graph

The com-Youtube graph contains 1,134,890 nodes and 2,987,624 edges. It contains friendship relations from Youtube. Looking at the benchmark times for this graph, we see a factor of around 2.5x speedup from our LAGraph implementation to the original implementation when we include/exclude d_{15}. However, when excluding both d_{15} and d_{12}, we see a very significant speedup.

This demonstrates that the vast majority of computation time is spent on computing these two graphlet frequency vectors.

3.2 The com-LiveJournal Graph

The com-LiveJournal graph contains 3,997,962 nodes and 34,681,189 edges. It contains friendship relations from LiveJournal, an online blogging community. This graph has an order of magnitude more edges than com-Youtube. We see a significant reduction in performance on this graph. However, except for when we include the computation of d_{15}, we see similar multipliers of around 2.5x from the original fglt implementation to ours in LAGraph.

3.3 LAGraph

The major result of this research will be the inclusion of the Fast Graphlet Transform into the LAGraph experimental branch [7]. We have demonstrated the applicability of GraphBLAS to the problem of computing graphlet frequency vectors using the Fast Graphlet Transform. However, there are improvements that can be made, both from the perspective of GraphBLAS' applicability to problems such as these, and in this experimental LAGraph implementation itself. To this we turn as we now conclude.

4. CONCLUSION

In general, the original implementation of Fast Graphlet Transform by the paper authors is around 2-3x faster than ours in LAGraph using GraphBLAS. The LAGraph implementation, though, has many advantages, including its readability and ease of implementation. Except for the two computations discussed which are not very well-suited to GraphBLAS, all graphlet counting computations described in the Fast Graphlet Transform paper [1] are extraordinarily straightforward to translate into a GraphBLAS implementation in LAGraph.

To conclude, we discuss briefly future directions of research. As mentioned, computing the graphlet frequency vectors d_{12} and d_{15} is not particularly well-suited to GraphBLAS.

In the case of d_{12}, the reason at a high level is that we want to compute A^2, and then reduce across its rows. In other words, we do not need the full matrix A^2. However, GraphBLAS has no way of knowing that our next computation is a reduction, and therefore must generate the full matrix. We avoid this now by manually splitting the matrix into blocks and reducing each block in parallel, but in general it is possible that a method of computing d_{12} exists that is more natural for GraphBLAS.

In the case of d_{15}, we seek to solve the problem of computing the K_4 frequency vector in a way that is natural for GraphBLAS. For now, we essentially replicate the method used in fglt [10] using `GxB_Iterator`. A trivial speedup can be achieved from our current implementation by parallelizing this iteration, but it is also possible that another way of viewing the problem would give us an approach more well-suited to GraphBLAS.

REFERENCES

[1] D. Floros, N. Pitsianis, and X. Sun, "Fast graphlet transform of sparse graphs," *CoRR*, vol. abs/2007.11111, 2020.

[2] D. B. West, *Introduction to Graph Theory*. Prentice Hall, 2 ed., September 2000.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[4] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2011.

[5] GraphBLAS contributors, "GraphBLAS." "https://graphblas.org", 2022. [Online].

[6] T. Davis, "SuiteSparse:GraphBLAS." "https://github.com/DrTimothyAldenDavis/GraphBLAS", 2022. [Online].

[7] G. Szárnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch, "Lagraph: Linear algebra, network analysis libraries, and the study of graph algorithms," 2021.

[8] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017.

[9] N. Pržulj, D. G. Corneil, and I. Jurisica, "Modeling interactome: scale-free or geometric?," *Bioinformatics*, vol. 20, pp. 3508–3515, 07 2004.

[10] D. Floros, N. Pitsianis, X. Sun, J. Barmparesos, and K. Kitsios, "FGlT." "https://github.com/fcdimitr/fglt", 2022. [Online].

[11] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection." "http://snap.stanford.edu/data", June 2014.

[12] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, Dec. 2011.